International Tables for Crystallography (2006). Vol. G, Chapter 5.6, pp. 544–556.

5.6. *CBFlib*: an ANSI C library for manipulating image data

By P. J. Ellis and H. J. Bernstein

5.6.1. Introduction

CBFlib is a library of ANSI C functions providing a simple mechanism for accessing crystallographic binary files (CBFs) and image-supporting CIF (imgCIF) files (see Chapters 2.3, 3.7 and 4.6). The *CBFlib* application programming interface (API) consists of a set of low-level functions and a set of high-level functions. The low-level functions are loosely based on the *CIFPARSE* (Tosic & Westbrook, 2000) API for mmCIF files. As in *CIFPARSE*, the low-level functions in *CBFlib* do not perform any semantic integrity checks and simply create, read, modify and write files conforming to the CIF syntax, with additional functionality for working with binary sections. These basic functions are independent of the details of the CBF/imgCIF dictionary and can be used to manage any CIF data set. In contrast, the high-level functions are based on the CBF/imgCIF dictionary and facilitate writing or reading commonly used entries to or from CBF and imgCIF data files.

External to a program, a CBF/imgCIF data set 'lives' in a file. Internally, when managed by *CBFlib*, a CBF or imgCIF data set has a simple tree structure pointed to by a 'handle' (Fig. 5.6.1.1). At the highest level are named data blocks. Each data block may contain a number of named categories. Within each category, the actual data entries are stored in tabular form with named columns and numbered rows. The numbers of rows in different columns of a given category are constrained by the software to be the same.

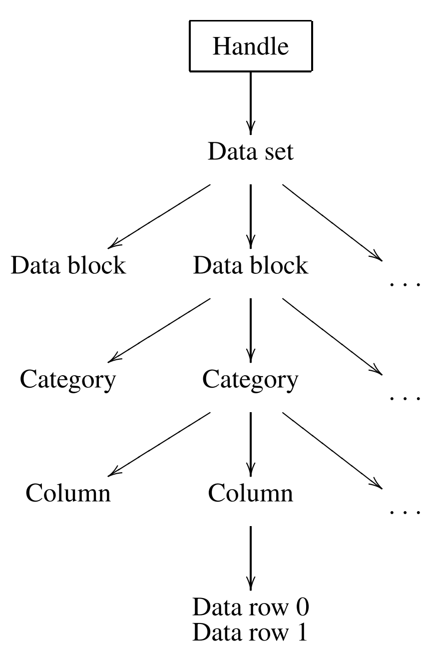

Fig. 5.6.1.1. Schematic data structure of a CBF or imgCIF data set accessed through a handle.

Affiliations: Paul J. Ellis, Stanford Linear Accelerator Center, 2575 Sand Hill Road, Menlo Park, CA 94025, USA; Herbert J. Bernstein, Department of Mathematics and Computer Science, Kramer Science Center, Dowling College, Idle Hour Blvd, Oakdale, NY 11769, USA.

Table 5.6.1.1. *Return values from CBFlib functions*

Error code	Hexadecimal value	Meaning
`CBF_FORMAT`	1	Invalid file format
`CBF_ALLOC`	2	Memory allocation failed
`CBF_ARGUMENT`	4	Invalid function argument
`CBF_ASCII`	8	Value is ASCII (not binary)
`CBF_BINARY`	10	Value is binary (not ASCII)
`CBF_BITCOUNT`	20	Expected number of bits does not match the actual number written
`CBF_ENDOFDATA`	40	End of the data reached before end of array
`CBF_FILECLOSE`	80	File close error
`CBF_FILEOPEN`	100	File open error
`CBF_FILEREAD`	200	File read error
`CBF_FILESEEK`	400	File seek error
`CBF_FILETELL`	800	File tell error
`CBF_FILEWRITE`	1000	File write error
`CBF_IDENTICAL`	2000	A data block with the new name already exists
`CBF_NOTFOUND`	4000	Data block, category, column or row does not exist
`CBF_OVERFLOW`	8000	Number read cannot fit into the destination argument; destination has been set to the nearest value
`CBF_UNDEFINED`	10000	Requested number not defined
`CBF_NOTIMPLEMENTED`	20000	Requested functionality is not yet implemented

CBFlib provides functions to create a corresponding data structure in memory; to copy a data set from an external file to the data structure or from the data structure to an external file; to navigate the tree; to scan, add and remove data blocks within data sets, categories (tables) within data blocks, and rows or columns within categories; to read or modify data entries; and finally to delete the structure from memory.

As is common in C programming, all functions return an integer equal to 0 for success or an error code for failure. The *CBFlib* error codes are given in Table 5.6.1.1.

CBFlib is thread-safe, re-entrant and able to operate on multiple data sets at a time. This means that the library maintains no static data and that the object to be operated on must be passed to each function. In *CBFlib*, this is accomplished by referring to each data set in memory with a unique handle of type `cbf_handle`. The handle maintains a link to the data structure in memory as well as the current location on the tree (data block, category, column and row). Before reading or creating a data set, the handle is created by calling the `cbf_make_handle` function. When the data set is no longer required, the resources associated with it can be freed using `cbf_free_handle`. Most functions in the library expect a handle as the first argument.

CBF binary data files and imgCIF ASCII data files may have one or more large images or other data sections as values for CIF tags. The focus of *CBFlib* is to handle large data sections efficiently.

The basic flow of an application reading CBF/imgCIF data with the low-level *CBFlib* functions is shown in Fig. 5.6.1.2.

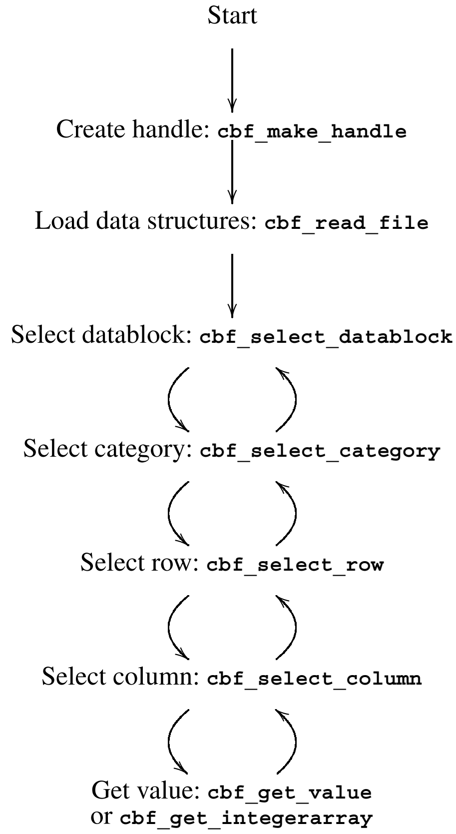

Fig. 5.6.1.2. Flow chart for a typical application reading CBF/imgCIF data.

The general approach to reading CBF/imgCIF data with *CBFlib* is to create an empty data structure with `cbf_make_handle`, load the data structures with `cbf_read_file` and then use nested loops to work through data blocks, categories, rows and columns in turn to extract values. Conceptually, all data values are held in the memory-resident data structures. In practice, however, only pointers to text fields with image data are held in memory. The data themselves remain on disk until explicitly referenced.

The basic flow of an application writing CBF/imgCIF data with the low-level *CBFlib* functions is shown in Fig. 5.6.1.3.

The general approach to writing CBF/imgCIF data with *CBFlib* is to create empty data structures with `cbf_make_handle` and load the data structures with nested loops, working through data blocks, categories, rows and columns in turn, to store values. The major difference from the nested loops used for reading is that empty columns are created before data are stored into the data structures row by row. Alternatively, the data could be stored column by column. Finally, the fully loaded memory data structures are written out with `cbf_write_file`. As with reading, text fields with image data are actually held on disk.

5.6.2. *CBFlib* function descriptions

All *CBFlib* functions have two common characteristics: (i) they return an integer equal to 0 for success or an error code for failure; (ii) any pointer argument for the result of an operation can be safely set to NULL. The error codes are given in Table 5.6.1.1.

CBFlib provides two low-level functions to create or destroy the structure used to hold a data set:

```
cbf_make_handle
cbf_free_handle
```

There are two functions to copy a data set from or into a file:

```
cbf_read_file
cbf_write_file
```

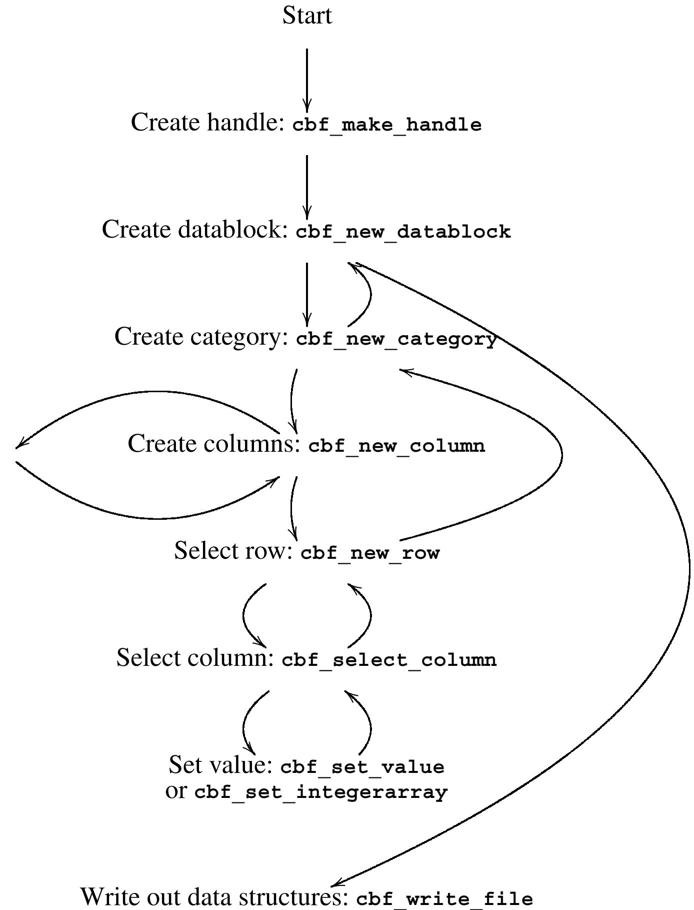

Fig. 5.6.1.3. Flow chart for an application writing CBF/imgCIF data.

The data structures 'behind' the handles retain pointers to current locations. This facilitates scanning through a CIF or CBF by data blocks, categories, rows and columns. The term '`rewind`' refers to setting the internal pointer for the type of item specified so that the first such item is pointed to.

In general, CIF does not permit duplication of the names of data blocks or category names. In practice, however, duplications do occur. *CBFlib* provides '`force`' variants of some functions to allow creation of duplicate names.

In *CBFlib*, the term '`set`' refers to changing the name of the currently specified item. The term '`reset`' refers to emptying a data block or category without deleting it. The term '`remove`' refers to deleting a data block, category, column or row. The terms '`select`' and '`next`' refer to finding the designated item by number, while the term '`find`' refers to finding the designated item by name.

CBFlib provides the following functions to manage data blocks and categories:

```
cbf_set_datablockname
     ⎧   new     ⎫
     ⎪ force_new ⎪
     ⎪   reset   ⎪
cbf_ ⎨  remove   ⎬ ⎧ _datablock ⎫
     ⎪  rewind   ⎪ ⎩ _category  ⎭
     ⎪  select   ⎪
     ⎪   next    ⎪
     ⎩   find    ⎭
cbf_reset_datablocks
cbf_count ⎧ _datablocks ⎫
          ⎩ _categories ⎭
cbf_ ⎧ datablock ⎫ _name
     ⎩ category  ⎭
```

The following functions manage columns and rows:

```
     ⎧  new    ⎫
     ⎪ remove  ⎪
     ⎪ rewind  ⎪
cbf_ ⎨  next   ⎬ { _column }
     ⎪  find   ⎪ { _row    }
     ⎪  count  ⎪
     ⎩ select  ⎭
cbf_column_name
cbf_row_number
cbf_ { insert } _row
     { delete }
cbf_find_nextrow
```

The following functions are provided to manage data values:

```
                ⎧ _value         ⎫
cbf_ { get }    ⎪ _integervalue  ⎪
     { set }    ⎨ _doublevalue   ⎬
                ⎩ _integerarray  ⎭
cbf_get_integerarrayparameters
```

Two macro definitions are provided to facilitate the handling of errors:

```
cbf_failnez
cbf_onfailnez
```

CBFlib also provides higher-level routines to simplify the management of complex CBF/imgCIF data sets:

```
cbf_read_template
                ⎧ _diffrn_id          ⎫
                ⎪ _crystal_id         ⎪
                ⎪ _wavelength         ⎪
                ⎪ _polarization       ⎪
                ⎪ _divergence         ⎪
cbf_ { get }    ⎨ _gain               ⎬
     { set }    ⎪ _overload           ⎪
                ⎪ _integration_time   ⎪
                ⎪ _time               ⎪
                ⎪ _date               ⎪
                ⎪ _image              ⎪
                ⎩ _axis_setting       ⎭
cbf_count_elements
cbf_get_element_id
cbf_set_current_time
cbf_get_image_size
cbf_ { construct } { _goniometer }
     {   free    } { _detector   }
cbf_get_rotation_ { axis  }
                  { range }
cbf_rotate_vector
cbf_get_reciprocal
         ⎧ beam_center        ⎫
         ⎪ detector_distance  ⎪
cbf_get_ ⎨ detector_normal    ⎬
         ⎪ pixel_coordinates  ⎪
         ⎪ pixel_normals      ⎪
         ⎩ pixel_area         ⎭
```

5.6.2.1. Low-level *CBFlib* functions

The prototypes for low-level *CBFlib* functions are defined in the header file cbf.h, which should be included in any program that uses *CBFlib*. As noted previously, every function returns an integer equal to 0 to indicate success or an error code on failure (Table 5.6.1.1).

The arguments to *CBFlib* functions are based on a view of a CBF/imgCIF data set as a tree (Fig. 5.6.1.1). The root of the tree is the data set and is identified by a handle that points to the data structures representing that tree. The main branches of the tree are the data blocks, identified by name or by number. Within each data block, the tree branches into categories, each of which branches into columns. Categories and columns also are identified by name or by number. Within each column is an array of values, the rows of which are identified by number. The current data block, category, column and row are stored in the data structures of a data set.

The following function descriptions include the formal parameters. When a '$*$' appears before a formal parameter, it is a pointer to the relevant value, rather than the actual value. The formal parameters for the low-level *CBFlib* functions are given in Table 5.6.2.1.

Table 5.6.2.1. *Formal parameters for low-level CBFlib functions*

`array`	Untyped array, typically holding a pointer to an image
`binary_id`	Integer identifier of a binary section
`categories`	Integer used for a count of categories
`category`	Integer ordinal of a category, counting from 0
`categoryname`	Character string; the name of a category
`ciforcbf`	Integer; selects the format in which the binary sections are written (CIF/CBF)
`column`	Integer ordinal of a column, counting from 0
`columnname`	Character string; the name of a column
`columns`	Integer count of columns in a category
`compression`	Integer designating the compression method used
`datablock`	Integer ordinal of a data block, counting from 0
`datablockname`	Character string; the name of a data block
`datablocks`	Integer count of data blocks in a CBF/imgCIF data set
`elements`	Number of elements in the array
`elements_read`	Pointer to the destination number of elements actually read
`elsigned`	Set to nonzero if the destination array elements are signed
`elsize`	Size in bytes of each array element
`elunsigned`	Pointer to an integer; set to 1 if the elements can be read as unsigned integers
`encoding`	Integer; selects the type of encoding used for binary sections and the type of line termination in imgCIF files
`file`	File descriptor
`handle`	CBF handle
`headers`	Integer; controls/selects the type of header in CBF binary sections and message digest generation
`maxelement`	Integer; largest element
`minelement`	Integer; smallest element
`number`	Integer or double value
`readable`	Integer; if nonzero: this file is random-access and readable, and can be used as a buffer
`row`	Integer; row ordinal
`rows`	Integer; row count
`value`	Integer or double value

Before working with a CBF (or CIF), it is necessary to create a handle. When work with the CBF is completed, the handle and associated data structures should be released:

```
int cbf_make_handle (cbf_handle *handle);
int cbf_free_handle (cbf_handle handle);
```

Normally, processing cannot continue if a handle is not created. Typical code to create a handle is:

```
#include "cbf.h"
cbf_handle cif;

if ( cbf_make_handle (&cif) ) {
   fprintf(stderr,
     "Failed to create handle for input_cif\n");
   exit(1);
 }
```

Table 5.6.2.2. *Values for headers in* `cbf_read_file`

MSG_DIGEST	Check that the digest of the binary section matches any header value. If the digests do not match, the call will return CBF_FORMAT. The evaluation and comparison is delayed (a 'lazy' evaluation) to ensure maximal processing efficiency. If an immediate evaluation is desired, see MSG_DIGESTNOW below.
MSG_DIGESTNOW	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 desired, see MSG_DIGEST above.
MSG_NODIGEST	Do not check the digest (default).

Once a handle has been created, the data structures can be loaded with all the information held in a CBF file:

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

Conceptually, all data values are associated with the handle at the `cbf_read_file` call. In practice, however, only the non-binary data are actually stored in memory. To work with potentially large binary sections most efficiently, these are skipped until explicitly referenced. For this reason, *file* must be a random-access file opened in binary mode [`fopen (..., "rb")`] and must not be closed by the calling program. *CBFlib* will call `fclose` when the file is no longer required.

The *headers* parameter controls the handling of any message digests embedded in the binary sections (Table 5.6.2.2). A *headers* value of `MSG_DIGEST` will cause the code to compare the digest of the binary section with any header message digest value. To maximize processing efficiency, this comparison will be delayed until the binary section is actually read into memory or copied (a 'lazy' evaluation). If immediate evaluation is required, use `MSG_DIGESTNOW`. In either case, if the digests do not match, the function in which the evaluation is taking place will return the error `CBF_FORMAT`. To ignore any digests, use the *headers* value `MSG_NODIGEST`.

The `cbf_write_file` call writes out the data associated with a CBF handle:

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

This call has several options controlling whether binary sections are written unencoded (CBF) or encoded in ASCII to conform to the CIF syntax (imgCIF), the type of headers in the binary sections, and the type of ASCII encoding and line termination used. The acceptable values for *ciforcbf* are `CIF` for ASCII-encoded binary sections or `CBF` for unencoded binary sections. The *headers* parameter (Table 5.6.2.3) can take the value `MIME_HEADERS` to select MIME-type binary section headers or `MIME_NOHEADERS` for simple ASCII headers. The value `MSG_DIGEST` will generate digests for validation of the binary data and the value `MSG_NODIGEST` will skip digest evaluation. The header and digest flags may be combined using the logical OR operator.

Similarly, there are several combinable flags for the parameter *encoding* (Table 5.6.2.4). `ENC_BASE64` selects BASE64 encoding, `ENC_QP` selects quoted-printable encoding, and `ENC_BASE8`, `ENC_BASE10` and `ENC_BASE16` select octal, decimal and hexadecimal, respectively. `ENC_FORWARD` maps bytes to words forward (1234) for BASE8, BASE10 or BASE16 encoding and `ENC_BACKWARD` maps bytes to words backward (4321). Finally, `ENC_CRTERM` terminates lines with carriage return (CR) and `ENC_LFTERM` terminates lines with line feed (LF) (thus `ENC_CRTERM|ENC_LFTERM` will use CR LF).

Table 5.6.2.3. *Values for headers in* `cbf_write_file`

Values may be combined bit-wise.

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

CBFlib maintains temporary storage on disk as necessary for files to be written, so that *file* does not have to be random-access. However, if it is random-access and readable, resources can be conserved by setting *readable* nonzero.

The remaining low-level functions are involved in navigating the tree structure, creating and deleting data blocks, categories and table columns and rows, and retrieving or modifying data values.

The navigation functions are:

```
int cbf_find_datablock (cbf_handle handle,
     const char *datablockname);
int cbf_find_category (cbf_handle handle,
     const char *categoryname);
int cbf_find_column (cbf_handle handle,
     const char *columnname);
int cbf_find_row (cbf_handle handle,
     const char *value);
int cbf_find_nextrow (cbf_handle handle,
     const char *value);

int cbf_select_datablock (cbf_handle handle,
     unsigned int datablock);
int cbf_select_category (cbf_handle handle,
     unsigned int category);
int cbf_select_column (cbf_handle handle,
     unsigned int column);
int cbf_select_row (cbf_handle handle,
     unsigned int row);

int cbf_rewind_datablock (cbf_handle handle);
int cbf_rewind_category (cbf_handle handle);
int cbf_rewind_column (cbf_handle handle);
int cbf_rewind_row (cbf_handle handle);

int cbf_next_datablock (cbf_handle handle);
int cbf_next_category (cbf_handle handle);
int cbf_next_column (cbf_handle handle);
int cbf_next_row (cbf_handle handle);
```

The function `cbf_find_datablock` selects the first data block with name *datablockname* as the current data block. Similarly, `cbf_find_category` selects the category within the current data block with name *categoryname* and `cbf_find_column` selects the corresponding column within the current category. The function `cbf_find_row` differs slightly in that it selects the first row in the current column with the corresponding *value* and `cbf_find_nextrow` selects the row with the corresponding value following the current row. Note that selecting a new data block makes the current category, column and row undefined and that selecting a new category similarly makes the column and row undefined. In contrast, repositioning by column does not change the current row and repositioning by row does not change the current column.

The remaining functions navigate on the basis of the order of the data blocks, categories, columns and rows. Thus, `cbf_select_datablock` selects data-block number *datablock*, counting from 0, `cbf_rewind_datablock` selects the first data

Table 5.6.2.4. *Values for encodings in* `cbf_write_file`

Values may be combined bit-wise.

`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`	For BASE8, BASE10 or BASE16 encoding, map bytes to words backward (4321) (default on big-endian machines)
`ENC_CRTERM`	Terminate lines with CR
`ENC_LFTERM`	Terminate lines with LF (default)

block and `cbf_next_datablock` selects the data block following the current data block.

All of these functions return `CBF_NOTFOUND` if the requested object does not exist.

The '`count`' functions evaluate the number of data blocks in the data set, the number of categories in the current data block and the number of columns or rows in the current category:

```
int cbf_count_datablocks (cbf_handle handle,
     unsigned int *datablocks);
int cbf_count_categories (cbf_handle handle,
     unsigned int *categories);
int cbf_count_columns (cbf_handle handle,
     unsigned int *columns);
int cbf_count_rows (cbf_handle handle,
     unsigned int *rows);
```

The '`name`' functions retrieve the current data block, category or column names:

```
int cbf_datablock_name (cbf_handle handle,
     const char **datablockname);
int cbf_set_datablockname (cbf_handle handle,
     const char *datablockname);
int cbf_category_name (cbf_handle handle,
     const char **categoryname);
```

As rows do not have names, the corresponding function is:

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

To create new entities within the tree, *CBFlib* provides the functions:

```
int cbf_new_datablock (cbf_handle handle,
     const char *datablockname);
int cbf_new_category (cbf_handle handle,
     const char *categoryname);
int cbf_new_column (cbf_handle handle,
     const char *columnname);
int cbf_new_row (cbf_handle handle);
int cbf_insert_row (cbf_handle handle,
     unsigned int row);
int cbf_force_new_datablock (cbf_handle handle,
     const char *datablockname);
int cbf_force_new_category (cbf_handle handle,
     const char *categoryname);
```

The '`new`' functions add a new data block within the data set, a new category in the current data block, or a new column or row within the current category, and make it the current data block, category, column or row, respectively. If the data block, category or column already exists, then the function simply makes it the current data block, category or column. The function `cbf_new_row` adds the row to the end of the current category. `cbf_insert_row` provides the ability to insert a row before ordinal *row*, starting from 0. The newly inserted row gets the row ordinal *row* and the row that originally had that ordinal and all rows with higher ordinals are pushed downwards.

In general, CIF does not permit duplication of the names of data blocks or categories. In practice, however, duplications do occur. *CBFlib* provides '`force`' variants to allow creation of duplicate data-block and category names. Because, in this case, the program analysing the resulting file can only distinguish the duplicates by ordinal, these variants are not recommended for general use.

The following functions are used to remove entities from the tree:

```
int cbf_remove_datablock (cbf_handle handle);
int cbf_remove_category (cbf_handle handle);
int cbf_remove_column (cbf_handle handle);
int cbf_remove_row (cbf_handle handle);
int cbf_delete_row (cbf_handle handle,
     unsigned int row);
int cbf_reset_datablocks (cbf_handle handle);
int cbf_reset_datablock (cbf_handle handle);
int cbf_reset_category (cbf_handle handle);
```

The basic '`remove`' functions delete the current data block, category, column or row. Note that removing a data block makes the current data block, category, column and row undefined; removing a category makes the current category, column and row undefined. Removing a column makes the current column undefined, but leaves the current row intact, and removing a row leaves the current column intact. The function `cbf_delete_row` is similar to `cbf_remove_row` except that it removes the specified row in the current category. If the current row is not the deleted row, then it will remain valid.

All the categories in all data blocks, all the categories in the current data block or all the entries in the current category may be removed using the '`reset`' functions.

When a column and row within a category have been selected, the entry value may be examined or modified:

```
int cbf_get_value (cbf_handle handle,
     const char **value);
int cbf_set_value (cbf_handle handle,
     const char *value);
int cbf_get_integervalue (cbf_handle handle,
     int *number);
int cbf_set_integervalue (cbf_handle handle,
     int number);
int cbf_get_doublevalue (cbf_handle handle,
     double *number);
int cbf_set_doublevalue (cbf_handle handle,
     const char *format;
int cbf_get_integerarrayparameters (cbf_handle handle,
     unsigned int *compression, size_t *elsize,
     size_t *elements, int *maxelement);
int cbf_get_integerarray (cbf_handle handle,
     int *binary_id, int elsigned,
     size_t *elements_read);
int cbf_set_integerarray (cbf_handle handle,
     unsigned int compression, void *array,
     size_t elements);
```

A value within a CBF/imgCIF data set may be a simple character string, an integer or real number, or an array of integers. The functions `cbf_get_value` and `cbf_set_value` provide the basic functionality for normal CIF values, retrieving and modifying the

Table 5.6.2.5. *Values for the parameter* `compression` *in* `cbf_get_integerarrayparameters` *and* `cbf_set_integerarray`

CBF_CANONICAL	Canonical-code compression (Section 5.6.3.1)
CBF_PACKED	*CCP4*-style packing (Section 5.6.3.2)
CBF_NONE	No compression

current entry as a string. The functions `cbf_get_integervalue` and `cbf_get_doublevalue` interpret the retrieved string as an integer or real value and the functions `cbf_set_integer` and `cbf_set_doublevalue` convert the *`number`* argument into a string before setting the entry.

The functions for working with binary sections are more complicated as they must take into account compression, array size and the variety of different integer types available on different systems: signed/unsigned and various sizes.

The function `cbf_get_integerarrayparameters` retrieves the parameters of the current, binary, entry. The *`compression`* argument is set to the compression type used (Table 5.6.2.5). At present, this may take one of three values: `CBF_CANONICAL`, for canonical-code compression (see Section 5.6.3.1 below); `CBF_PACKED`, for *CCP4*-style packing (see Section 5.6.3.2 below); or `CBF_NONE`, for no compression. [*Note*: `CBF_NONE` is by far the slowest scheme 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 *`binary_id`* value is a unique integer identifier for each binary section, *`elsize`* is the size in bytes of the array entries, *`elsigned`* and *`elunsigned`* are nonzero if the array can be read as unsigned or signed, respectively, *`elements`* is the number of entries in the array, and *`minelement`* and *`maxelement`* are the lowest and highest elements. If a destination argument is too small to hold a value, it will be set to the nearest value and the function will return `CBF_OVERFLOW`. If the current entry is not binary, `cbf_get_integerarrayparameters` will return `CBF_ASCII`.

`cbf_get_integerarray` reads the current binary entry into an integer array. The parameter *`array`* points to an array of elements interpreted as integers. Each element in the array is signed if *`elsigned`* is nonzero and unsigned otherwise, and each element occupies *`elsize`* bytes. The argument *`elements_read`* is set to the number of elements actually obtained. If the binary section does not contain sufficient entries to fill the array, the function returns `CBF_ENDOFDATA`. As before, the function will return `CBF_OVERFLOW` on overflow and `CBF_ASCII` if the entry is not binary.

`cbf_set_integerarray` sets the current binary or ASCII entry to the binary value of an integer array. As before, the acceptable values for compression are `CBF_PACKED`, `CBF_CANONICAL` and `CBF_NONE`. Each binary section should be given a unique integer identifier *`binary_id`*.

Two macros are provided to facilitate processing and propagation of error returns: one to return from the current function immediately and one to execute a given command first:

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

If the symbol `CBFDEBUG` is defined, alternative definitions that print out the error number as given in Table 5.6.1.1 are used:

```
#define cbf_failnez(x) \
{int err; err = (x); \
  if (err) { fprintf (stderr, \
    "\nCBFlib error %d in \"x\"\n", \
    err); return err; }}

#define cbf_onfailnez(x,c) \
{int err; err = (x); \
  if (err) { fprintf (stderr, \
    "\nCBFlib error %d in \"x\"\n", \
     err); \
     { c; } return err; }}
```

5.6.2.2. High-level *CBFlib* functions

The high-level *CBFlib* functions provide a level of abstraction above the CIF file structure and their prototypes are defined in the header file cbf_simple.h. Most of these functions simply use the low-level routines to navigate the CBF/imgCIF structure and read and modify data entries, and consequently expect a `cbf_handle` argument. There are also, however, additional sets of functions used to analyse the geometry of the goniometer and detector. These functions use additional handles of type `cbf_goniometer` and `cbf_detector`, respectively. All functions return the same error codes as the low-level functions do. The function return values are given in Table 5.6.1.1. The formal parameters for the high-level *CBFlib* functions are given in Table 5.6.2.6.

5.6.2.3. General high-level functions

The general high-level functions use the low-level routines to accomplish common tasks with a single call.

The first of these is used to facilitate the preparation of the complex CBF/imgCIF header structure:

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

`cbf_read_template` simply reads the CBF/imgCIF file *`file`* into the data structure associated with the given handle and selects the first data block. It is typically used to read a template – an imgCIF file populated with data entries, but without any binary sections, into which experimental information can then be inserted. Template files are discussed further in Section 5.6.4 below.

The value of **`_diffrn_radiation_wavelength.wavelength`** can be retrieved or set. The functions

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

operate on the categories DIFFRN_RADIATION and DIFFRN_RADIATION_WAVELENGTH. The wavelength is found indirectly. The value of **`_diffrn_radiation.wavelength_id`** is retrieved and used to find a matching row in the DIFFRN_RADIATION_WAVELENGTH category, from which the value of **`_diffrn_radiation_wavelength.wavelength`** is obtained.

The value of the ratio of the intensities of the polarization components **`_diffrn_radiation.polarizn_source_ratio`** and the value of the angle **`_diffrn_radiation.polarizn_source_norm`** between the normal to the polarization plane and the laboratory *Y* axis can be retrieved or set. The functions

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

operate on the DIFFRN_RADIATION category.

Table 5.6.2.6. *Formal parameters for high-level CBFlib functions*

`array`	Pointer to image array data
`axis_id`	Axis ID
`center1`	Displacement along the slow axis
`center2`	Displacement along the fast axis
`compression`	Compression type
`coordinate1`	*x* component
`coordinate2`	*y* component
`coordinate3`	*z* component
`crystal_id`	ASCII crystal ID
`day`	Timestamp day (1–31)
`detector`	Detector handle
`diffrn_id`	ASCII diffraction ID
`distance`	Distance
`div_x_source`	Value of **`_diffrn_radiation.div_x_source`**
`div_x_y_source`	Value of **`_diffrn_radiation.div_x_y_source`**
`div_y_source`	Value of **`_diffrn_radiation.div_y_source`**
`element_id`	ASCII element ID
`element_number`	Detector element counting from 0
`elements`	Count of elements
`elsigned`	Set to nonzero if the destination array elements are signed
`elsize`	Size in bytes of each destination array element
`file`	File descriptor
`gain`	Detector gain in counts per photon
`gain_esd`	Gain e.s.d. value
`goniometer`	Goniometer handle
`handle`	CBF handle
`hour`	Timestamp hour (0–23)
`increment`	Increment value
`index1`	Slow index
`index2`	Fast index
`initial1`	*x* component of the initial vector
`initial2`	*y* component of the initial vector
`initial3`	*z* component of the initial vector
`minute`	Timestamp minute (0–59)
`month`	Timestamp month (1–12)
`ndim1`	Slow array dimension
`ndim2`	Fast array dimension
`normal1`	*x* component of the normal vector
`normal2`	*y* component of the normal vector
`normal3`	*z* component of the normal vector
`overload`	Overload value
`polarizn_source_norm`	Polarization normal
`polarizn_source_ratio`	Polarization ratio
`precision`	Timestamp precision in seconds
`projected_area`	Apparent area in mm^2
`ratio`	Goniometer setting (0 = beginning of exposure, 1 = end)
`real1`	*x* component of the real-space vector
`real2`	*y* component of the real-space vector
`real3`	*z* component of the real-space vector
`reciprocal1`	*x* component of the reciprocal-space vector
`reciprocal2`	*y* component of the reciprocal-space vector
`reciprocal3`	*z* component of the reciprocal-space vector
`reserved`	Unused; any value other than 0 is invalid
`second`	Timestamp second (0–60.0)
`start`	Start value
`time`	Timestamp in seconds since 1 January 1970 or integration time in seconds
`timezone`	Time zone difference from universal time in minutes or `CBF_NOTIMEZONE`
`vector1`	*x* component of the rotation axis
`vector2`	*y* component of the rotation axis
`vector3`	*z* component of the rotation axis
`wavelength`	Wavelength in Å
`year`	Pointer to the destination timestamp year

The values of the divergence parameters, represented by the data names **`_diffrn_radiation.div_x_source`**, **`*.div_y_source`** and **`*.div_x_y_source`**, can be retrieved or set. The functions

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

operate on the DIFFRN_RADIATION category.

The values of **`_diffrn.id`** and **`_diffrn.crystal_id`** can be retrieved or set:

```
int cbf_get_diffrn_id (cbf_handle handle,
     const char **diffrn_id);
int cbf_set_diffrn_id (cbf_handle handle,
     const char *diffrn_id);
int cbf_get_crystal_id (cbf_handle handle,
     const char **crystal_id);
int cbf_set_crystal_id (cbf_handle handle,
     const char *crystal_id);
```

Changing **`_diffrn.id`** also modifies the corresponding **`*.diffrn_id`** entries in the DIFFRN_SOURCE, DIFFRN_RADIATION, DIFFRN_DETECTOR and DIFFRN_MEASUREMENT categories.

The starting value and increment of an axis may be retrieved or set:

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

The `cbf_set_axis_setting` call is used during the creation of a CBF/imgCIF file to store the goniometer settings and rotation. The `cbf_get_axis_setting` is not generally useful when interpreting a file as there are no standard identifiers and the arrangement of the experimental axes is not consistent. Much more useful are the goniometer geometry functions described below.

The number of detector elements can be retrieved:

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

This is the number of rows in the DIFFRN_DETECTOR_ELEMENT category. For each element, counting from 0, the detector identifier (the **`*.detector_id`** entry) can be retrieved and the gain and overload values in the ARRAY_INTENSITIES category retrieved or set:

```
int cbf_get_element_id (cbf_handle handle,
     unsigned int element_number,
     const char **element_id);
int cbf_get_gain (cbf_handle handle,
     unsigned int element_number,
     double *gain, double *gain_esd);
int cbf_set_gain (cbf_handle handle,
     unsigned int element_number, double gain,
     double gain_esd);
int cbf_get_overload (cbf_handle handle,
     unsigned int element_number, double *overload);
int cbf_set_overload (cbf_handle handle,
     unsigned int element_number, double overload);
```

For each element, counting from 0, the values of the parameters of the detector can be retrieved and some can be set. The value of **`_diffrn_detector_element.id`** is retrieved as *`element_id`*. The value of **`_diffrn_data_frame.array_id`** can be retrieved as *`array_name`*. The values of **`_array_intensities.gain`** and **`_array_intensities.gain_esd`** are retrieved as *`gain`* and *gain_esd*. The value of **_array_intensities.overload** can be retrieved or set as *overload*. The value of **_diffrn_scan_frame.integration_time** can be retrieved or set as *integration_time*.

Timestamp calls operate on the DATE entry in the DIFFRN_SCAN_FRAME category:

```
int cbf_get_timestamp (cbf_handle handle,
     unsigned int reserved, double *time,
     int *timezone);
int cbf_set_timestamp (cbf_handle handle,
     unsigned int reserved, double time,
     int timezone, double precision);
int cbf_get_datestamp (cbf_handle handle,
     unsigned int reserved, int *year, int *month,
     int *day, int *hour, int *minute, double *second,
     int *timezone);
int cbf_set_datestamp (cbf_handle handle,
     unsigned int reserved, int year, int month,
     int day, int hour, int minute, double second,
     int timezone, double precision);
int cbf_set_current_timestamp (cbf_handle handle,
     unsigned int reserved, int timezone)
```

`cbf_get_timestamp` and `cbf_set_timestamp` measure time in seconds since 1 January 1970. `cbf_get_datestamp` and `cbf_set_datestamp` work in terms of individual *year*, *month*, *day*, *hour*, *minute* and *second*. The optional collection time zone, *timezone*, is the difference from universal time in minutes; *precision* is the fraction, in seconds, to which the time will be recorded. `cbf_set_current_timestamp` sets the collection timestamp from the current time, to the nearest second.

Also in the DIFFRN_SCAN_FRAME category is the integration time of the image:

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

Finally, these functions include routines for working with binary images:

```
int cbf_get_image_size (cbf_handle handle,
     unsigned int reserved,
     unsigned int element_number,
     size_t *ndim1, size_t *ndim2);
int cbf_get_image (cbf_handle handle,
     unsigned int reserved,
     unsigned int element_number,
     void *array, size_t elsize, int elsign,
     size_t ndim1, size_t ndim2);
int cbf_set_image (cbf_handle handle,
     unsigned int reserved,
     unsigned int element_number,
     unsigned int compression, void *array,
     size_t elsize, int elsign, size_t ndim1,
     size_t ndim2);
```

`cbf_get_image_size` retrieves the dimensions of detector element *element_number* from the ARRAY_STRUCTURE_LIST category, setting *ndim1* and *ndim2* to the slow and fast array dimensions, respectively. These dimensions can be used to allocate memory before calling `cbf_get_image`. `cbf_get_image` reads the image data from detector element *element_number* into a signed or unsigned integer array of size *ndim1 * ndim2* and `cbf_set_image` associates image data with a detector element. As in the description of the integer array functions, the compression argument can currently take one of three values: `CBF_CANONICAL`, for canonical-code compression (see Section 5.6.3.1); `CBF_PACKED`, for *CCP4*-style packing (see Section 5.6.3.2); or `CBF_NONE`, for no compression.

5.6.2.4. Goniometer geometry functions

A CBF/imgCIF file includes a geometric description of the goniometer used to orient the sample during the experiment. Practical use of this information, however, is not trivial as it involves combining data from several categories and analysing in three dimensions the nested axes in which the description is framed (see Section 3.7.3 for a discussion of the axis system). *CBFlib* provides six functions to facilitate this task:

```
int cbf_construct_goniometer (cbf_handle handle,
     cbf_goniometer *goniometer);
int cbf_free_goniometer (cbf_goniometer goniometer);
int cbf_get_rotation_axis (cbf_goniometer goniometer,
     unsigned int reserved, double *vector1,
     double *vector2, double *vector3);
int cbf_get_rotation_range (cbf_goniometer goniometer,
     unsigned int reserved, double *start,
     double *increment);
int cbf_rotate_vector (cbf_goniometer goniometer,
     unsigned int reserved, double ratio,
     double initial1, double initial2, double initial3,
     double *final1, double *final2, double *final3);
int cbf_get_reciprocal (cbf_goniometer goniometer,
     unsigned int reserved, double ratio,
     double wavelength, double real1, double real2,
     double real3, double *reciprocal1,
     double *reciprocal2, double *reciprocal3);
```

`cbf_construct_goniometer` uses the data in the categories DIFFRN_MEASUREMENT, DIFFRN_MEASUREMENT_AXIS, AXIS, DIFFRN_SCAN_FRAME_AXIS and DIFFRN_SCAN_AXIS to construct a geometric representation of the goniometer and initializes the `cbf_goniometer` handle, *goniometer*. `cbf_free_goniometer` frees the goniometer structure. `cbf_get_ rotation_axis` and `cbf_get_rotation_range` get the normalized rotation vector, and the starting value and increment of the first rotating axis of the goniometer, respectively. The `cbf_rotate_vector` call applies the goniometer axis rotation to the given initial vector, with the *ratio* value specifying the goniometer setting from 0.0 at the beginning of the exposure to 1.0 at the end, irrespective of the actual rotation range. Finally, `cbf_get_reciprocal` transforms the given real-space vector (*real1, real2, real3*) to the corresponding reciprocal-space vector (*reciprocal1, reciprocal2, reciprocal3*). As before, the transform corresponds to the goniometer initial position with a *ratio* of 0.0 and the goniometer final position with a *ratio* of 1.0.

5.6.2.5. Detector geometry functions

In a similar manner, a CBF/imgCIF file includes a description of the surface of each detector and the arrangement of the pixels in space. *CBFlib* provides eight functions for analysing this description:

```
int cbf_construct_detector (cbf_handle handle,
     cbf_detector *detector,
     unsigned int element_number);
int cbf_free_detector (cbf_detector detector);
int cbf_get_beam_center (cbf_detector detector,
     double *index1, double *index2,
     double *center1, double *center2);
int cbf_get_detector_distance (cbf_detector detector,
     double *distance);
```

```
int cbf_get_detector_normal (cbf_detector detector,
     double *normal1, double *normal2,
     double *normal3);
int cbf_get_pixel_coordinates (cbf_detector detector,
     double index1, double index2,
     double *coordinate1, double *coordinate2,
     double *coordinate3);
int cbf_get_pixel_normal (cbf_detector detector,
     double index1, double index2,
     double *normal1, double *normal2,
     double *normal3);
int cbf_get_pixel_area (cbf_detector detector,
     double index1, double index2,
     double *area, double *projected_area);
```

`cbf_construct_detector` uses data from the categories DIFFRN, DIFFRN_DETECTOR, DIFFRN_DETECTOR_ELEMENT, DIFFRN_DETECTOR_AXIS, AXIS, ARRAY_STRUCTURE_LIST and ARRAY_STRUCTURE_LIST_AXIS to construct a geometric representation of detector element *element_number* and initializes the `cbf_detector` handle, *detector*. `cbf_free_detector` frees the detector structure; `cbf_get_beam_center` calculates the location at which the beam intersects the detector surface, either in terms of the pixel indices (*index1, index2*) along the slow and fast detector axes, respectively, or the displacement in millimetres along the slow and fast axes (*center1, center2*); `cbf_get_detector_distance` and `cbf_get_detector_normal` calculate the distance of the sample from the plane of the detector surface and the normal vector of the detector at pixel $(0, 0)$, respectively; `cbf_get_pixel_coordinates`, `cbf_get_pixel_normal` and `cbf_get_pixel_area` calculate the coordinates, normal vector, and area and apparent area as viewed from the sample position of the pixel with the given indices, respectively.

5.6.3. Compression schemes

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

(i) an entropy-encoding scheme using canonical coding;

(ii) 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.

5.6.3.1. 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 itself.

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} to $2^{n-1} - 1$, one for a stop code and one for each of the $maxbits - n$ indirect codes, where n is chosen at compression 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.* (1997).

Table 5.6.3.1. *Structure of compressed data using the canonical-code scheme*

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

The structure of the compressed data is described in Table 5.6.3.1.

5.6.3.2. *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 to 2. Bits 3 to 5 encode the number of bits in each error. The data structure is summarized in Table 5.6.3.2.

Table 5.6.3.2. *Structure of compressed data using the CCP4-style scheme*

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

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

5.6.4. Sample templates

The construction of CBF/imgCIF files can be simplified using templates. A template is itself an imgCIF file populated with data entries but without any binary sections. This file is normally associated with a CBF handle using the `cbf_read_template` call and provides a framework into which images and other experiment-specific data may be entered.

Fig. 5.6.4.1 is a sample template for an ADSC Quantum 4 detector (ADSC, 1997) with a κ-geometry diffractometer at Stanford Synchrotron Radiation Laboratory (SSRL) beamline 1-5.

The template for a MAR345 image plate detector (MAR Research, 1997) is almost identical. The major differences are in the size of the array (2300 $\times$ 2300 *versus* 2304 $\times$ 2304), the parameters for the CCD elements and the geometry of the elements. Therefore a few of the values in the AXIS, ARRAY_STRUCTURE_LIST, ARRAY_STRUCTURE_LIST_AXIS and ARRAY_INTENSITIES categories are different, as listed in Fig. 5.6.4.2.

```
###CBF: VERSION 1.1

data_image_1

# category DIFFRN
loop_
_diffrn.id
_diffrn.crystal_id
 DIFFRN_ID DIFFRN_CRYSTAL_ID

# category DIFFRN_SOURCE
loop_
_diffrn_source.diffrn_id
_diffrn_source.source
_diffrn_source.current
_diffrn_source.type
 DIFFRN_ID synchrotron 100.0 'SSRL beamline 1-5'

# category DIFFRN_DETECTOR_ELEMENT
loop_
_diffrn_detector_element.id
_diffrn_detector_element.detector_id
 ELEMENT1 ADSCQ4

# category DIFFRN_RADIATION
loop_
_diffrn_radiation.diffrn_id
_diffrn_radiation.wavelength_id
_diffrn_radiation.probe
_diffrn_radiation.monochromator
_diffrn_radiation.polarizn_source_ratio
_diffrn_radiation.polarizn_source_norm
_diffrn_radiation.div_x_source
_diffrn_radiation.div_y_source
_diffrn_radiation.div_x_y_source
_diffrn_radiation.collimation
 DIFFRN_ID WAVELENGTH1 x-ray 'Si 111' 0.8 0.0 0.08
    0.01 0.00 '0.20 mm x 0.20 mm'

# category DIFFRN_RADIATION_WAVELENGTH
loop_
_diffrn_radiation_wavelength.id
_diffrn_radiation_wavelength.wavelength
_diffrn_radiation_wavelength.wt
 WAVELENGTH1 0.98 1.0

# category DIFFRN_DETECTOR
loop_
_diffrn_detector.diffrn_id
_diffrn_detector.id
_diffrn_detector.type
_diffrn_detector.details
_diffrn_detector.number_of_axes
 DIFFRN_ID ADSCQ4 'ADSC QUANTUM4' 'slow mode' 4

 # category DIFFRN_DETECTOR_AXIS
loop_
_diffrn_detector_axis.detector_id
_diffrn_detector_axis.axis_id
 ADSCQ4 DETECTOR_X
 ADSCQ4 DETECTOR_Y
 ADSCQ4 DETECTOR_Z
 ADSCQ4 DETECTOR_PITCH

# category DIFFRN_DATA_FRAME
loop_
_diffrn_data_frame.id
_diffrn_data_frame.detector_element_id
_diffrn_data_frame.array_id
_diffrn_data_frame.binary_id
 FRAME1 ELEMENT1 ARRAY1 1
```

Fig. 5.6.4.1. Template imgCIF for use with an ADSC Quantum 4 detector.

```
# category DIFFRN_MEASUREMENT
loop_
_diffrn_measurement.diffrn_id
_diffrn_measurement.id
_diffrn_measurement.number_of_axes
_diffrn_measurement.method
_diffrn_measurement.details
 DIFFRN_ID GONIOMETER 3 rotation
; i0=1.000 i1=1.000 i2=1.000 ib=1.000 beamstop=20 mm
  0% attenuation
;


# category DIFFRN_MEASUREMENT_AXIS
loop_
_diffrn_measurement_axis.measurement_id
_diffrn_measurement_axis.axis_id
 GONIOMETER GONIOMETER_PHI
 GONIOMETER GONIOMETER_KAPPA
 GONIOMETER GONIOMETER_OMEGA


# category DIFFRN_SCAN
loop_
_diffrn_scan.id
_diffrn_scan.frame_id_start
_diffrn_scan.frame_id_end
_diffrn_scan.frames
 SCAN1 FRAME1 FRAME1 1


# category DIFFRN_SCAN_AXIS
loop_
_diffrn_scan_axis.scan_id
_diffrn_scan_axis.axis_id
_diffrn_scan_axis.angle_start
_diffrn_scan_axis.angle_range
_diffrn_scan_axis.angle_increment
_diffrn_scan_axis.displacement_start
_diffrn_scan_axis.displacement_range
_diffrn_scan_axis.displacement_increment
 SCAN1 GONIOMETER_OMEGA 0.0 0.0 0.0 0.0 0.0 0.0
 SCAN1 GONIOMETER_KAPPA 0.0 0.0 0.0 0.0 0.0 0.0
 SCAN1 GONIOMETER_PHI   0.0 0.0 0.0 0.0 0.0 0.0
 SCAN1 DETECTOR_Z       0.0 0.0 0.0 0.0 0.0 0.0
 SCAN1 DETECTOR_Y       0.0 0.0 0.0 0.0 0.0 0.0
 SCAN1 DETECTOR_X       0.0 0.0 0.0 0.0 0.0 0.0
 SCAN1 DETECTOR_PITCH   0.0 0.0 0.0 0.0 0.0 0.0

# category DIFFRN_SCAN_FRAME
loop_
_diffrn_scan_frame.frame_id
_diffrn_scan_frame.frame_number
_diffrn_scan_frame.integration_time
_diffrn_scan_frame.scan_id
_diffrn_scan_frame.date
 FRAME1 1 0.0 SCAN1 1997-12-04T10:23:48

# category DIFFRN_SCAN_FRAME_AXIS
loop_
_diffrn_scan_frame_axis.frame_id
_diffrn_scan_frame_axis.axis_id
_diffrn_scan_frame_axis.angle
_diffrn_scan_frame_axis.displacement
 FRAME1 GONIOMETER_OMEGA 0.0 0.0
 FRAME1 GONIOMETER_KAPPA 0.0 0.0
 FRAME1 GONIOMETER_PHI   0.0 0.0
 FRAME1 DETECTOR_Z       0.0 0.0
 FRAME1 DETECTOR_Y       0.0 0.0
 FRAME1 DETECTOR_X       0.0 0.0
 FRAME1 DETECTOR_PITCH   0.0 0.0
```

Fig. 5.6.4.1. (*cont.*)

```
# category AXIS
loop_
_axis.id
_axis.type
_axis.equipment
_axis.depends_on
_axis.vector[1] _axis.vector[2] _axis.vector[3]
_axis.offset[1] _axis.offset[2] _axis.offset[3]
 GONIOMETER_OMEGA rotation    goniometer
   . 1 0 0 . . .
 GONIOMETER_KAPPA rotation    goniometer
   GONIOMETER_OMEGA 0.64279 0 0.76604 . . .
 GONIOMETER_PHI   rotation    goniometer
   GONIOMETER_KAPPA 1 0 0 . . .
 SOURCE           general     source   . 0 0 1 . . .
 GRAVITY          general     gravity  . 0 -1 0 . . .
 DETECTOR_Z       translation detector
   . 0 0 -1 0 0 0
 DETECTOR_Y       translation detector
   DETECTOR_Z 0 1 0 0 0 0
 DETECTOR_X       translation detector
   DETECTOR_Y 1 0 0 0 0 0
 DETECTOR_PITCH   rotation    detector
   DETECTOR_X 0 1 0 0 0 0
 ELEMENT_X        translation detector
   DETECTOR_PITCH 1 0 0 -94.0032 94.0032 0
 ELEMENT_Y        translation detector
   ELEMENT_X 0 1 0 0 0 0

# category ARRAY_STRUCTURE_LIST
loop_
_array_structure_list.array_id
_array_structure_list.index
_array_structure_list.dimension
_array_structure_list.precedence
_array_structure_list.direction
_array_structure_list.axis_set_id
 ARRAY1 1 2304 1 increasing ELEMENT_X
 ARRAY1 2 2304 2 increasing ELEMENT_Y

# category ARRAY_STRUCTURE_LIST_AXIS
loop_
_array_structure_list_axis.axis_set_id
_array_structure_list_axis.axis_id
_array_structure_list_axis.displacement
_array_structure_list_axis.displacement_increment
 ELEMENT_X ELEMENT_X 0.0408 0.0816
 ELEMENT_Y ELEMENT_Y -0.0408 -0.0816

# category ARRAY_INTENSITIES
loop_
_array_intensities.array_id
_array_intensities.binary_id
_array_intensities.linearity
_array_intensities.gain
_array_intensities.gain_esd
_array_intensities.overload
_array_intensities.undefined_value
 ARRAY1 1 linear 0.23 0.03 65000 0


 # category ARRAY_STRUCTURE
loop_
_array_structure.id
_array_structure.encoding_type
_array_structure.compression_type
_array_structure.byte_order
 ARRAY1 "signed 32-bit integer" packed little_endian
```

Fig. 5.6.4.1. (*cont.*)

```
# category ARRAY_DATA
loop_
_array_data.array_id
_array_data.binary_id
_array_data.data
 ARRAY1 1 ?
```

Fig. 5.6.4.1. (*cont.*)

```
loop_
_axis.id
_axis.type
_axis.equipment
_axis.depends_on
_axis.vector[1] _axis.vector[2] _axis.vector[3]
_axis.offset[1] _axis.offset[2] _axis.offset[3]
 GONIOMETER_OMEGA rotation    goniometer
   . 1 0 0 . . .
 GONIOMETER_KAPPA rotation    goniometer
   GONIOMETER_OMEGA 0.64279 0 0.76604 . . .
 GONIOMETER_PHI   rotation    goniometer
   GONIOMETER_KAPPA 1 0 0 . . .
 SOURCE           general     source
   . 0 0 1 . . .
 GRAVITY          general     gravity
   . 0 -1 0 . . .
 DETECTOR_Z       translation detector
   . 0 0 -1 0 0 0
 DETECTOR_Y       translation detector
   DETECTOR_Z 0 1 0 0 0 0
 DETECTOR_X       translation detector
   DETECTOR_Y 1 0 0 0 0 0
 DETECTOR_PITCH   rotation    detector
   DETECTOR_X 0 1 0 0 0 0
 ELEMENT_X        translation detector
   DETECTOR_PITCH 1 0 0 -172.5 172.5 0
 ELEMENT_Y        translation detector
   ELEMENT_X 0 1 0 0 0 0

loop_
_array_structure_list.array_id
_array_structure_list.index
_array_structure_list.precedence
_array_structure_list.direction
_array_structure_list.axis_set_id
 ARRAY1 1 2300 1 increasing ELEMENT_X
 ARRAY1 2 2300 2 increasing ELEMENT_Y

 loop_
_array_structure_list_axis.axis_set_id
_array_structure_list_axis.axis_id
_array_structure_list_axis.displacement
_array_structure_list_axis.displacement_increment
 ELEMENT_X ELEMENT_X 0.075 0.150
 ELEMENT_Y ELEMENT_Y -0.075 -0.150

 loop_
_array_intensities.array_id
_array_intensities.binary_id
_array_intensities.linearity
_array_intensities.gain
_array_intensities.gain_esd
_array_intensities.overload
_array_intensities.undefined_value
 ARRAY1 1 linear 1.15 0.2 240000 0
```

Fig. 5.6.4.2. Part of the template file for a MAR345 detector. Values that differ from those in Fig. 5.6.4.1 are underlined.

```
  /* Read and modify the template */

cbf_failnez (cbf_make_handle (\&cbf))
in = fopen (template_name, "rb");

if (!in)
  exit (4);
cbf_failnez (cbf_read_template (cbf, in))

  /* Wavelength */

wavelength =
  img_get_number (img, "WAVELENGTH");
if (wavelength)
  cbf_failnez (
    cbf_set_wavelength (cbf, wavelength))

  /* Distance */

distance = img_get_number (img, "DISTANCE");
cbf_failnez (
  cbf_set_axis_setting (
    cbf, 0, "DETECTOR_Z", distance, 0))

  /* Oscillation start and range */

axis = img_get_field (img, "OSCILLATION AXIS");

if (!axis)

  axis = "PHI";

osc_start = img_get_number (img, axis);
osc_range = img_get_number (img,
   "OSCILLATION RANGE");
cbf_failnez (
  cbf_set_axis_setting (
    cbf, 0, "GONIOMETER_PHI",
      osc_start, osc_range))
```

Fig. 5.6.5.1. An extract of program code for the conversion of image data to CBF format.

5.6.5. Example programs

The *CBFlib* API comes with example programs and sample templates. The example program *convert_image* reads either of the templates described in Section 5.6.4, template_adscquantum4_2304x2304.cbf and template_mar345_2300x2300.cbf, and uses the higher-level *CBFlib* routines to convert an image file to a CBF.

The example programs *makecbf* and *img2cif* read an image file from a MAR300, MAR345 or ADSC CCD detector and then use *CBFlib* to convert that image to CBF format (*makecbf*) or to either imgCIF or CBF format (*img2cif*). *makecbf* writes the CBF-format image to disk, reads it in again, and then compares it with the original. *img2cif* just writes the desired file without a verification pass. *makecbf* works only from files on disk, so that random input/output can be used. *img2cif* includes code to process files from stdin and to stdout. The example program *cif2cbf* copies an ASCII CIF to a binary CBF/imgCIF file.

5.6.5.1. *convert_image*

The program *convert_image* takes two arguments, `imagefile` and `cbffile`. These are the primary input and output. The detector type is extracted from the image file, converted to lower case and used to construct the name of a template CBF to use for the copy. The template file name is of the form *template_name_columnsxrows*. The program comes with img.c and img.h to read image files. It uses the `img_...` routines to extract data and metadata from the image and uses the higher-level *CBFlib* routines to populate the template. Fig. 5.6.5.1 is a portion of the code that reads in the template and inserts the wavelength, the distance to the detector, and the oscillation start and range.

```
/* Make the _diffrn_frame_data category */
cbf_failnez(
  cbf_new_category         /* create the category */
    (cbf, "diffrn_frame_data"))
cbf_failnez(
  cbf_new_column           /* create the column   */
    (cbf, "id"))
cbf_failnez(
  cbf_set_value            /* add data            */
    (cbf, "frame_1"))
cbf_failnez(
  cbf_new_column           /* create next column  */
    (cbf, "detector_element_id"))
cbf_failnez(
  cbf_set_integervalue     /* add data            */
    (cbf, 1))
cbf_failnez(
  cbf_new_column           /* create the column   */
    (cbf, "detector_id"))
cbf_failnez(
  cbf_set_value            /* add data            */
    (cbf, detector_id))
cbf_failnez(
  cbf_new_column           /* create next column  */
   (cbf, "array_id"))
cbf_failnez(
  cbf_set_value            /* add data            */
    (cbf, "image_1"))
cbf_failnez(
  cbf_new_column           /* create next column  */
    (cbf, "binary_id"))
cbf_failnez(
  cbf_set_integervalue     /* add data            */
    (cbf, 1))
```

Fig. 5.6.5.2. Code fragment to illustrate the creation of the DIFFRN_FRAME_DATA category.

5.6.5.2. *makecbf*

makecbf is a good example of how to use many of the lower-level *CBFlib* functions. An example MAR345 image can be found at http://smb.slac.stanford.edu/~ellis/CBF_examples/mar345/mb_example_070.cbf. To run *makecbf* with the example image, type:

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

The typical code fragment from makecbf.c shown in Fig. 5.6.5.2 creates the DIFFRN_FRAME_DATA category.

The program *img2cif* is an extended version of *makecbf* that allows the user to choose the details of the format of the output CBF. *img2cif* has the following command-line interface:

```
img2cif [-i input_image] [-o output_cif]
        [-c  {p[acked]|c[anonical]|[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]
```

-i takes the name of the input image file in MAR300, MAR345 or ADSC CCD detector format. If no `input_image` file is specified or is given as '-', an image is copied from stdin to a temporary file. *-o* takes the name of 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* specifies the compression scheme (packed, canonical or none; the default is packed). *-m* selects MIME (Freed &

```
cbf_failnez (cbf_count_datablocks(cif, &blocks))
for (blocknum = 0; blocknum < blocks;  blocknum++)
{ /* start of copy loop */
  cbf_failnez (
    cbf_select_datablock(cif, blocknum))
  cbf_failnez (
    cbf_datablock_name(cif, &datablock_name))
  cbf_failnez (
    cbf_force_new_datablock(cbf, datablock_name))
  if ( !cbf_rewind_category(cif) ) {
    cbf_failnez (
      cbf_count_categories(cif, &categories))
    for (catnum = 0; catnum < categories; catnum++) {
      cbf_select_category(cif, catnum);
      cbf_category_name(cif,&category_name);
      cbf_force_new_category(cbf, category_name);
      cbf_count_rows(cif,&rows);
      cbf_count_columns(cif,&columns);
      /*  Transfer the column names from cif to cbf */
      if ( ! cbf_rewind_column(cif) ) {
      do {
        cbf_failnez(
          cbf_column_name(cif, &column_name))
        cbf_failnez(
          cbf_new_column(cbf, column_name))
      } while ( ! cbf_next_column(cif) );
      cbf_rewind_column(cif);
      cbf_rewind_row(cif);
      }
      /* Transfer the rows from cif to cbf */
      for (rownum = 0; rownum < rows; rownum++) {
        cbf_failnez (
          cbf_select_row(cif, rownum))
        cbf_failnez ( cbf_new_row(cbf))
        cbf_rewind_column(cif);
        for (colnum = 0; colnum < columns; colnum++) {
          cbf_failnez (cbf_select_column(cif, colnum))
          if ( ! cbf_get_value(cif, &value) ) {
            cbf_failnez (
              cbf_select_column(cbf, colnum))
            cbf_failnez ( cbf_set_value(cbf, value))
          } else {
            void * array;
            int binary_id, elsigned, elunsigned;
            size_t elements,elements_read, elsize;
            int minelement, maxelement;
            unsigned int cifcompression;

            cbf_failnez (
              cbf_get_integerarrayparameters(cif,
              &cifcompression, &binary_id, &elsize,
              &elsigned, &elunsigned, &elements,
              &minelement, &maxelement))
            if (array=malloc(elsize*elements)) {
              cbf_failnez (
                cbf_select_column(cbf,colnum))
              cbf_failnez (
                cbf_get_integerarray(
                cif, &binary_id, array, elsize,
                elsigned, elements, &elements_read))
              cbf_failnez(
                cbf_set_integerarray(
                cbf, compression, binary_id, array,
                elsize, elsigned, elements))
              free(array);
            } else {
              fprintf(stderr,
              "\nFailed to allocate memory %d bytes",
                elsize*elements);
               exit(1);
            }
          }
        }
      }
    }
  }
```

Fig. 5.6.5.3. Listing of the main code fragment in the program *cif2cbf* for conversion of an ASCII CIF to a CBF file.

Borenstein, 1996) headers within binary data value text fields. The default is headers for CIFs, no headers for CBFs. *-d* specifies the use of digests. The default is the MD5 digest (Rivest, 1992) when MIME headers are selected. *-e* specifies one of the standard MIME encodings: BASE64 (the default) or quoted-printable; or a non-standard decimal, hexadecimal or octal encoding for an ASCII CIF; or 'none' for a binary CBF. *-b* specifies the direction of mapping of bytes into words for decimal, hexadecimal or octal output, marked by '>' for forwards or '<' for backwards as the second character of each line of output, and in # comment lines. The default is backwards.

5.6.5.3. *cif2cbf*

The test program *cif2cbf* (Fig. 5.6.5.3) uses the same command-line options as *img2cif*, but accepts either a CIF or a CBF as input instead of an image file. The heart of the code is a series of nested loops. The outermost loop scans through all the data blocks. The next innermost loop scans through all the categories within each data block. The first of the two next innermost loops scans through the columns to copy the column headings. Then the second of these two loops scans through the rows. Finally, the innermost loop scans through the columns for each row.

We are grateful to Frances C. Bernstein for her helpful comments and suggestions.

References

ADSC (1997). *Quantum 4R specifications.* Poway, CA, USA: Area Detector Systems Corporation. http://www.adsc-xray.com/Q4techspecs.html.

Freed, N. & Borenstein, N. (1996). *Multipurpose Internet Mail Extensions (MIME) part one: format of internet message bodies.* RFC 2045. Internet Engineering Task Force. http://www.ietf.org/rfc/rfc2045.txt.

MAR Research (1997). *The mar345 image plate detector.* Norderstedt: MAR Research GmbH. http://www.marresearch.com/products.mar345.html.

Moffat, A., Bell, T. C. & Witten, I. H. (1997). *Lossless compression for text and images. Int. J. High Speed Electron. Syst.* **8**, 179–231.

Rivest, R. (1992). *The MD5 message-digest algorithm.* RFC 1321. Internet Engineering Task Force. http://www.ietf.org/rfc/rfc1321.txt.

Tosic, O. & Westbrook, J. D. (2000). *CIFParse. A library of access tools for mmCIF*. Reference guide. http://sw-tools.pdb.org/apps/CIFPARSE-OBJ/cifparse/index.html.