The following functions manage columns and rows:

```
cbf_ { new | remove | rewind | next | find | count | select } { _column | _row }
cbf_column_name
cbf_row_number
cbf_ { insert | delete } _row
cbf_find_nextrow
```

The following functions are provided to manage data values:

```
cbf_ { get | set } { _value | _integervalue | _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
cbf_ { get | set } { _diffrn_id | _crystal_id | _wavelength | _polarization | _divergence | _gain | _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 | free } { _goniometer | _detector }
cbf_get_rotation_ { axis | range }
cbf_rotate_vector
cbf_get_reciprocal
cbf_get_ { beam_center | detector_distance | 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).

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

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.

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.