International Tables for Crystallography (2006). Vol. G, Section 5.6.2, pp. 545–552.

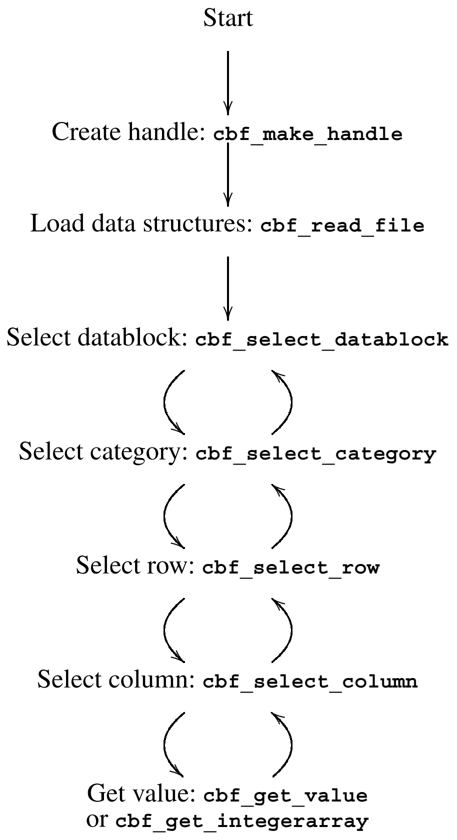
Start
Create handle: cbf_make_handle
Load data structures: cbf_read_file
Select datablock: cbf_select_datablock
Select category: cbf_select_category
Select row: cbf_select_row
Select column: cbf_select_column
Get value: cbf_get_value
or cbf_get_integerarray

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

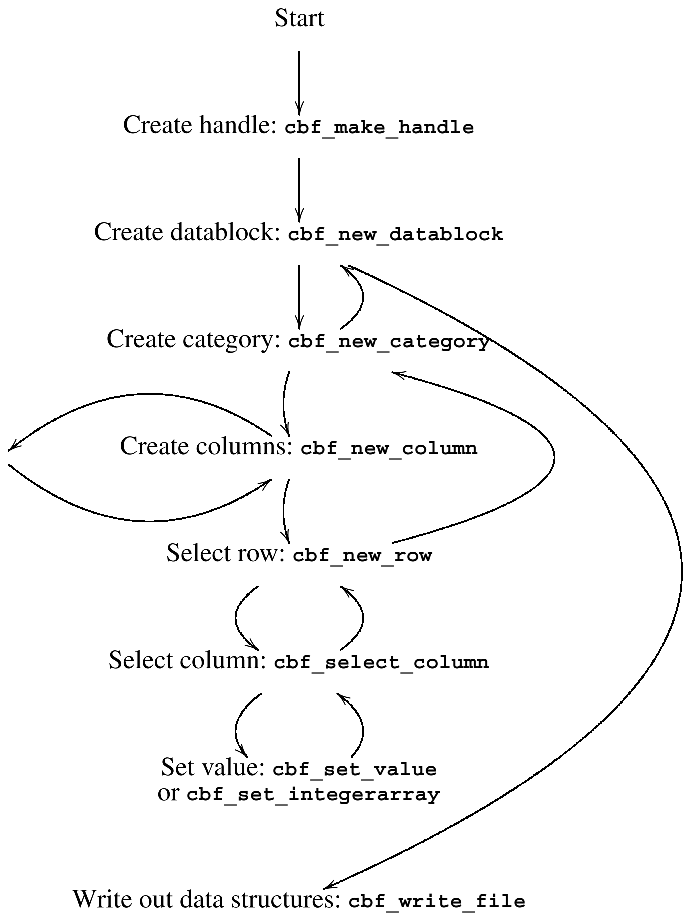
Start
Create handle: cbf_make_handle
Create datablock: cbf_new_datablock
Create category: cbf_new_category
Create columns: cbf_new_column
Select row: cbf_new_row
Select column: cbf_select_column
Set value: cbf_set_value
or cbf_set_integerarray
Write out data structures: cbf_write_file

Fig. 5.6.1.3. Flow chart for an application writing 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
```

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`

$$\texttt{cbf\_}\left\{\begin{array}{c}\texttt{new}\\\texttt{force\_new}\\\texttt{reset}\\\texttt{remove}\\\texttt{rewind}\\\texttt{select}\\\texttt{next}\\\texttt{find}\end{array}\right\}\left\{\begin{array}{c}\texttt{\_datablock}\\\texttt{\_category}\end{array}\right\}$$

`cbf_reset_datablocks`

$$\texttt{cbf\_count}\left\{\begin{array}{c}\texttt{\_datablocks}\\\texttt{\_categories}\end{array}\right\}$$

$$\texttt{cbf\_}\left\{\begin{array}{c}\texttt{datablock}\\\texttt{category}\end{array}\right\}\texttt{\_name}$$

The following functions manage columns and rows:

$$\texttt{cbf\_}\left\{\begin{array}{c}\texttt{new}\\\texttt{remove}\\\texttt{rewind}\\\texttt{next}\\\texttt{find}\\\texttt{count}\\\texttt{select}\end{array}\right\}\left\{\begin{array}{c}\texttt{\_column}\\\texttt{\_row}\end{array}\right\}$$

`cbf_column_name`
`cbf_row_number`

$$\texttt{cbf\_}\left\{\begin{array}{c}\texttt{insert}\\\texttt{delete}\end{array}\right\}\texttt{\_row}$$

`cbf_find_nextrow`

The following functions are provided to manage data values:

$$\texttt{cbf\_}\left\{\begin{array}{c}\texttt{get}\\\texttt{set}\end{array}\right\}\left\{\begin{array}{l}\texttt{\_value}\\\texttt{\_integervalue}\\\texttt{\_doublevalue}\\\texttt{\_integerarray}\end{array}\right\}$$

`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`

$$\texttt{cbf\_}\left\{\begin{array}{c}\texttt{get}\\\texttt{set}\end{array}\right\}\left\{\begin{array}{l}\texttt{\_diffrn\_id}\\\texttt{\_crystal\_id}\\\texttt{\_wavelength}\\\texttt{\_polarization}\\\texttt{\_divergence}\\\texttt{\_gain}\\\texttt{\_overload}\\\texttt{\_integration\_time}\\\texttt{\_time}\\\texttt{\_date}\\\texttt{\_image}\\\texttt{\_axis\_setting}\end{array}\right\}$$

`cbf_count_elements`
`cbf_get_element_id`
`cbf_set_current_time`
`cbf_get_image_size`

$$\texttt{cbf\_}\left\{\begin{array}{c}\texttt{construct}\\\texttt{free}\end{array}\right\}\left\{\begin{array}{l}\texttt{\_goniometer}\\\texttt{\_detector}\end{array}\right\}$$

$$\texttt{cbf\_get\_rotation\_}\left\{\begin{array}{c}\texttt{axis}\\\texttt{range}\end{array}\right\}$$

`cbf_rotate_vector`
`cbf_get_reciprocal`

$$\texttt{cbf\_get\_}\left\{\begin{array}{l}\texttt{beam\_center}\\\texttt{detector\_distance}\\\texttt{detector\_normal}\\\texttt{pixel\_coordinates}\\\texttt{pixel\_normals}\\\texttt{pixel\_area}\end{array}\right\}$$

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.

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

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

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.

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 × 2300 *versus* 2304 × 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.