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);
 }
```