PyCBF

A python binding to the CBFlib library

Jon P. Wright
Anyone who wishes to contribute, please do!

Started Dec 12, 2005, already it is June 30, 2021

Abstract

Area detectors at synchrotron facilities can result in huge amounts of data being generated very rapidly. The IUCr (International Union of Crystallography) has devised a standard file format for storing and annotating such data, in order that it might be more easily interchanged and exploited. A c library which gives access to this file format has been developed by Paul Ellis and Herbert Bernstein (Version 0.7.4, http://www.bernstein-plus-sons.com/software/CBF/). In this document a python interface is developed using the SWIG (http://www.swig.org) package in order to give the author easy access to binary cif files.

Contents

1 Introduction **2**

2 Installation prerequisites **3**

3 Generating the c interface - the SWIG file **3**
3.1 Exceptions 5
3.2 Exceptions 10

4 Docstrings **10**

5 Wrappers **10**

6 Building python extensions - the setup file **10**

7 Building and testing the resulting package **11**

8 Debugging compiled extensions **12**

9 Things which are currently missing **13**

10 Testing **13**
10.1 Read a file based on cif2cbf.c 13
10.2 Try to test the goniometer and detector 15
10.3 Test cases for the generics 15
10.4 Version of pycbf$_{t}est1withwritelogicadded$ 17
10.5 Processing of XFEL axes 18

11 Worked example 1 : xmas beamline + mar ccd detector at the ESRF **18**
11.1 Reading marccd headers 19
11.2 Writing out cif files for fit2d/xmas 21
11.3 A template cif file for the xmas beamline 23

Index of file names

"linux.sh" Defined by 12a.
"makeflatascii.py" Defined by 12b.
"make_pycbf.py" Defined by 10.
"py2setup_py.m4" Defined by 11a.
"pycbf.i" Defined by 5b, 9.
"pycbf_test1.py" Defined by 14.
"pycbf_test2.py" Defined by 15a.
"pycbf_test3.py" Defined by 15b.
"pycbf_test4.py" Defined by 17.
"pycbf_testfelaxes.py" Defined by 18.
"win32.bat" Defined by 11b.
"xmas/readmarheader.py" Defined by 20.
"xmas/xmasheaders.py" Defined by 22.
"xmas/xmas_cif_template.cif" Defined by 24.

Index of macro names

⟨ Constants used for compression 3a ⟩ Referenced in 9.
⟨ Constants used for encoding 4b ⟩ Referenced in 9.
⟨ Constants used for headers 3b ⟩ Referenced in 9.
⟨ Constants used to control CIF parsing 4a ⟩ Referenced in 9.
⟨ Exception handling 5a ⟩ Referenced in 9.

Things to do

- Write test code to test each and every function for good and bad args etc

1 Introduction

The CBFlib library (version 0.7.4) is written in the C language, offering C (and C++) programmers a convenient interface to such files. The current author uses a different language (python) from day to day and so a python interface was desired. After a short attempt to make a quick and dirty SWIG interface it was decided that in the long run it would be better to write a proper interface for python.

All of the functions in the library return an integer reflecting error status. Usually these integers seem to be zero, and a non-zero return value appears to mean an error occurred. Actual return values are returned via pointers in argument lists. In order to simplify the authors life (as a user) all of those integers have been made to disappear if they are zero, and cause an “exception” to be generated if they are not zero. This solution might not be the best thing to do, and it can always be changed where the return value is intended to normally be used.

Actual return values which were passed back via pointer arguments are now just passed back as (perhaps multiple) return values. We must look out for INOUT arguments, none seem to have been found yet, but there might be exceptions. The author has a vague suspicion that python functions generally do not modify their arguments, but this might be wrong.

The library appears to define (at least) three objects. The one we started on was the cbf_handle_struct defined in cbf.h. Many of the functions have their first argument as a pointer to one of these structures. Therefore we make this structure an object and then everything which uses it as first argument is a member function for that object.

In order to pass image data back and forth there is a difficulty that python seems to lack a good way to represent large arrays. The standard library offers an ”array” object which claims to efficiently hold homogenous numerical data. Sadly this seems to be limited to one-dimensional arrays. The builtin string object can hold binary data and this was chosen as the way to pass the actual binary back and forth between python and CBFlib. Unfortunately this means the binary data are pretty useless when they arrive on the python side, so helper functions are provided to convert the data to a python (standard library) 1D array and also to a ”Numeric” array or a ”Numarray” array. The latter two are popular extension modules for manipulating large arrays.

2 Installation prerequisites

The document you are reading was generated from a nuweb source file. This is something very similar to latex with a few extensions for writing out source code files. As such it keeps together the whole package in a single file and makes it easier to write documentation. You will need a to obtain the preprocessing tool nuweb (perhaps from http://nuweb.sourceforge.net) in order to build from scratch with the file pycbf.w. Preproccessed output is hopefully also available to you. We do not recommend editing the SWIG generated wrappers!!

Only python version 2.4 has been targetted originally (other versions?) so that you will probably want to have that version of python installed.

We are building binary extensions, so you also need a working c compiler. The compiler used by the author was gcc (for both windows and unix) with the mingw version under windows.

Finally, you need a copy of swig (from www.swig.org) in order to (re)generate the c wrappers.

In case all that sounds scary, then fear not, it is likely that a single download for windows will just work with the right version of python. Unix systems come with many of those things available anyway.

3 Generating the c interface - the SWIG file

Essentially the swig file starts by saying what to include to build the wrappers, and then goes on to define the python interface for each function we want to call.

The library appears to define at least three "objects"; a CBF handle, a cbf_goniometer and a cbf_detector. We will attempt to map these onto python classes.

FIXME - decide whether introduce a "binary array" class with converters to more common representations?

All of the functions in the library appear to return 0 on success and a meaningful error code on failure. We try to propagate that error code across the language barrier via exceptions.

So the SWIG file will start off by including the header files needed for compilation. Note the defintion of constants to be passed as arguments in calls in the form pycbf.CONSTANTNAME

⟨ *Constants used for compression* 3a ⟩ ≡

```
// The actual wrappers

// Constants needed from header files

  /* Constants used for compression */

#define CBF_INTEGER     0x0010  /* Uncompressed integer               */
#define CBF_FLOAT       0x0020  /* Uncompressed IEEE floating-point   */
#define CBF_CANONICAL   0x0050  /* Canonical compression              */
#define CBF_PACKED      0x0060  /* Packed compression                 */
#define CBF_PACKED_V2   0x0090  /* CCP4 Packed (JPA) compression V2   */
#define CBF_BYTE_OFFSET 0x0070  /* Byte Offset Compression            */
#define CBF_PREDICTOR   0x0080  /* Predictor_Huffman Compression      */
#define CBF_NONE        0x0040  /* No compression flag                */
#define CBF_COMPRESSION_MASK  \
                        0x00FF  /* Mask to separate compression
                                   type from flags              */
#define CBF_FLAG_MASK   0x0F00  /* Mask to separate flags from
                                   compression type             */
#define CBF_UNCORRELATED_SECTIONS \
                        0x0100  /* Flag for uncorrelated sections     */
#define CBF_FLAT_IMAGE  0x0200  /* Flag for flat (linear) images      */
#define CBF_NO_EXPAND   0x0400  /* Flag to try not to expand          */
```

◇

Fragment referenced in 9.

⟨ *Constants used for headers* 3b ⟩ ≡

```
  /* Constants used for headers */

#define PLAIN_HEADERS   0x0001  /* Use plain ASCII headers           */
#define MIME_HEADERS    0x0002  /* Use MIME headers                  */
#define MSG_NODIGEST    0x0004  /* Do not check message digests      */
#define MSG_DIGEST      0x0008  /* Check message digests             */
#define MSG_DIGESTNOW   0x0010  /* Check message digests immediately  */
#define MSG_DIGESTWARN  0x0020  /* Warn on message digests immediately*/
#define PAD_1K          0x0020  /* Pad binaries with 1023 0's        */
#define PAD_2K          0x0040  /* Pad binaries with 2047 0's        */
#define PAD_4K          0x0080  /* Pad binaries with 4095 0's        */
◇
```

Fragment referenced in 9.

⟨ *Constants used to control CIF parsing* 4a ⟩ ≡

```
  /* Constants used to control CIF parsing */

#define CBF_PARSE_BRC   0x0100  /* PARSE DDLm/CIF2 brace {,...}            */
#define CBF_PARSE_PRN   0x0200  /* PARSE DDLm parens     (,...)            */
#define CBF_PARSE_BKT   0x0400  /* PARSE DDLm brackets   [,...]            */
#define CBF_PARSE_BRACKETS \
                        0x0700  /* PARSE ALL brackets                      */
#define CBF_PARSE_TQ    0x0800  /* PARSE treble quotes """..."""" and '''...'''      */
#define CBF_PARSE_CIF2_DELIMS  \
                        0x1000  /* Do not scan past an unescaped close quote
                                   do not accept {} , : " ' in non-delimited
                                   strings'{ */
#define CBF_PARSE_DDLm  0x0700  /* For DDLm parse (), [], {}               */
#define CBF_PARSE_CIF2  0x1F00  /* For CIF2 parse {}, treble quotes,
                                   stop on unescaped close quotes          */
#define CBF_PARSE_DEFINES      \
                        0x2000  /* Recognize DEFINE_name           */


#define CBF_PARSE_WIDE      0x4000  /* PARSE wide files                        */

#define CBF_PARSE_UTF8      0x10000 /* PARSE UTF-8                             */

#define HDR_DEFAULT (MIME_HEADERS | MSG_NODIGEST)

#define MIME_NOHEADERS  PLAIN_HEADERS

  /* CBF vs CIF */

#define CBF             0x0000  /* Use simple binary sections        */
#define CIF             0x0001  /* Use MIME-encoded binary sections  */
◇
```

Fragment referenced in 9.

⟨ *Constants used for encoding* 4b ⟩ ≡

```
  /* Constants used for encoding */

#define ENC_NONE        0x0001  /* Use BINARY encoding               */
#define ENC_BASE64      0x0002  /* Use BASE64 encoding               */
#define ENC_BASE32K     0x0004  /* Use X-BASE32K encoding            */
```

```
#define ENC_QP          0x0008  /* Use QUOTED-PRINTABLE encoding       */
#define ENC_BASE10      0x0010  /* Use BASE10 encoding                 */
#define ENC_BASE16      0x0020  /* Use BASE16 encoding                 */
#define ENC_BASE8       0x0040  /* Use BASE8  encoding                 */
#define ENC_FORWARD     0x0080  /* Map bytes to words forward (1234)   */
#define ENC_BACKWARD    0x0100  /* Map bytes to words backward (4321)  */
#define ENC_CRTERM      0x0200  /* Terminate lines with CR             */
#define ENC_LFTERM      0x0400  /* Terminate lines with LF             */

#define ENC_DEFAULT (ENC_BASE64 | ENC_LFTERM | ENC_FORWARD)
◇
```

Fragment referenced in 9.

3.1 Exceptions

We attempt to catch the errors and pass them back to python as exceptions. This could still do with a little work to propagage back the calls causing the errors.

Currently there are two global constants defined, called error_message and error_status. These are filled out when an error occurred, converting the numerical error value into something the author can read.

There is an implicit assumption that if the library is used correctly you will not normally get exceptions. This should be addressed further in areas like file opening, proper python exceptions should be returned.

See the section on exception handling in pycbf.i, above.

Currently you get a meaningful string back. Should perhaps look into defining these as python exception classes? In any case - the SWIG exception handling is defined via the following. It could have retained the old style if(status = action) but then harder to see what to return...

⟨ *Exception handling* 5a ⟩ ≡

```
// Exception handling

  /* Convenience definitions for functions returning error codes */
%exception {
   error_status=0;
   $action
   if (error_status){
     get_error_message();
     PyErr_SetString(PyExc_Exception,error_message);
     return NULL;
   }
}

/* Retain notation from cbf lib but pass on as python exception */

#define cbf_failnez(x) {(error_status = x);}

/* printf("Called \"x\", status %d\n",error_status);} */

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

Fragment referenced in 9.

"pycbf.i" 5b≡

```
/* File: pycbf.i */

// Indicate that we want to generate a module call pycbf
%module pycbf

%pythoncode %{
__author__ = "Jon Wright <wright@esrf.fr>"
__date__ = "14 Dec 2005"
__version__ = "CBFlib 0.9"
__credits__ = """Paul Ellis and Herbert Bernstein for the excellent CBFlib!"""
__doc__=""" pycbf - python bindings to the CBFlib library

 A library for reading and writing ImageCIF and CBF files
 which store area detector images for crystallography.

 This work is a derivative of the CBFlib version 0.7.7 library
 by  Paul J. Ellis of Stanford Synchrotron Radiation Laboratory
 and Herbert J. Bernstein of Bernstein + Sons
 See:
   http://www.bernstein-plus-sons.com/software/CBF/

 Licensing is GPL based, see:
   http://www.bernstein-plus-sons.com/software/CBF/doc/CBFlib_NOTICES.html

 These bindings were automatically generated by SWIG, and the
 input to SWIG was automatically generated by a python script.
 We very strongly recommend you do not attempt to edit them
 by hand!



 Copyright (C) 2007    Jonathan Wright
                       ESRF, Grenoble, France
                email: wright@esrf.fr

  Revised, August 2010  Herbert J. Bernstein
    Add defines from CBFlib 0.9.1

"""
%}


// Used later to pass back binary data
%include "cstring.i"

// Attempt to autogenerate what SWIG thinks the call looks like

// Typemaps are a SWIG mechanism for many things, not least multiple
// return values
%include "typemaps.i"

// Arrays are needed
%include "carrays.i"
%array_class(double, doubleArray)
%array_class(int, intArray)
%array_class(short, shortArray)
%array_class(long, longArray)

// Following the SWIG 1.3 documentation at
// http://www.swig.org/Doc1.3/Python.html
// section 31.9.5, we map sequences of
// PyFloat, PyLong and PyInt to
```

```
// C arrays of double, long and int
//
// But with the strict checking of being a float
// commented out to allow automatic conversions
%{
static int convert_darray(PyObject *input, double *ptr, int size) {
  int i;
  if (!PySequence_Check(input)) {
      PyErr_SetString(PyExc_TypeError,"Expecting a sequence");
      return 0;
  }
  if (PyObject_Length(input) != size) {
      PyErr_SetString(PyExc_ValueError,"Sequence size mismatch");
      return 0;
  }
  for (i =0; i < size; i++) {
      PyObject *o = PySequence_GetItem(input,i);
     /*if (!PyFloat_Check(o)) {

         Py_XDECREF(o);
         PyErr_SetString(PyExc_ValueError,"Expecting a sequence of floats");
         return 0;
      }*/
      ptr[i] = PyFloat_AsDouble(o);
      Py_DECREF(o);
  }
  return 1;
}
%}

%typemap(in) double [ANY](double temp[$1_dim0]) {
    if ($input == Py_None) $1 = NULL;
    else
    if (!convert_darray($input,temp,$1_dim0)) {
      return NULL;
    }
    $1 = &temp[0];
}

%{
    static long convert_larray(PyObject *input, long *ptr, int size) {
        int i;
        if (!PySequence_Check(input)) {
            PyErr_SetString(PyExc_TypeError,"Expecting a sequence");
            return 0;
        }
        if (PyObject_Length(input) != size) {
            PyErr_SetString(PyExc_ValueError,"Sequence size mismatch");
            return 0;
        }
        for (i =0; i < size; i++) {
            PyObject *o = PySequence_GetItem(input,i);
            /*if (!PyLong_Check(o)) {
                Py_XDECREF(o);
                PyErr_SetString(PyExc_ValueError,"Expecting a sequence of long integers");
                return 0;
            }*/
            ptr[i] = PyLong_AsLong(o);
            Py_DECREF(o);
        }
        return 1;
    }
%}
```

```
%typemap(in) long [ANY](long temp[$1_dim0]) {
    if (!convert_larray($input,temp,$1_dim0)) {
        return NULL;
    }
    $1 = &temp[0];
}

%{
    static int convert_iarray(PyObject *input, int *ptr, int size) {
        int i;
        if (!PySequence_Check(input)) {
            PyErr_SetString(PyExc_TypeError,"Expecting a sequence");
            return 0;
        }
        if (PyObject_Length(input) != size) {
            PyErr_SetString(PyExc_ValueError,"Sequence size mismatch");
            return 0;
        }
        for (i =0; i < size; i++) {
            PyObject *o = PySequence_GetItem(input,i);
            /*if (!PyInt_Check(o)) {
                Py_XDECREF(o);
                PyErr_SetString(PyExc_ValueError,"Expecting a sequence of long integers");
                return 0;
            }*/
            ptr[i] = (int)PyInt_AsLong(o);
            Py_DECREF(o);
        }
        return 1;
    }
%}

%typemap(in) int [ANY](int temp[$1_dim0]) {
    if (!convert_iarray($input,temp,$1_dim0)) {
        return NULL;
    }
    $1 = &temp[0];
}


%{  // Here is the c code needed to compile the wrappers, but not
    // to be wrapped

#include "../include/cbf.h"
#include "../include/cbf_simple.h"

// Helper functions to generate error message


static int error_status = 0;
static char error_message1[17] ;
static char error_message[1042] ; // hope that is long enough

/* prototype */
void get_error_message(void);

void get_error_message(){
  sprintf(error_message1,"%s","CBFlib Error(s):");
  if (error_status & CBF_FORMAT        )
    sprintf(error_message,"%s %s",error_message1,"CBF_FORMAT       ");
  if (error_status & CBF_ALLOC         )
    sprintf(error_message,"%s %s",error_message1,"CBF_ALLOC        ");
```

```
  if (error_status & CBF_ARGUMENT      )
    sprintf(error_message,"%s %s",error_message1,"CBF_ARGUMENT     ");
  if (error_status & CBF_ASCII         )
    sprintf(error_message,"%s %s",error_message1,"CBF_ASCII        ");
  if (error_status & CBF_BINARY        )
    sprintf(error_message,"%s %s",error_message1,"CBF_BINARY       ");
  if (error_status & CBF_BITCOUNT      )
    sprintf(error_message,"%s %s",error_message1,"CBF_BITCOUNT     ");
  if (error_status & CBF_ENDOFDATA     )
    sprintf(error_message,"%s %s",error_message1,"CBF_ENDOFDATA    ");
  if (error_status & CBF_FILECLOSE     )
    sprintf(error_message,"%s %s",error_message1,"CBF_FILECLOSE    ");
  if (error_status & CBF_FILEOPEN      )
    sprintf(error_message,"%s %s",error_message1,"CBF_FILEOPEN     ");
  if (error_status & CBF_FILEREAD      )
    sprintf(error_message,"%s %s",error_message1,"CBF_FILEREAD     ");
  if (error_status & CBF_FILESEEK      )
    sprintf(error_message,"%s %s",error_message1,"CBF_FILESEEK     ");
  if (error_status & CBF_FILETELL      )
    sprintf(error_message,"%s %s",error_message1,"CBF_FILETELL     ");
  if (error_status & CBF_FILEWRITE     )
    sprintf(error_message,"%s %s",error_message1,"CBF_FILEWRITE    ");
  if (error_status & CBF_IDENTICAL     )
    sprintf(error_message,"%s %s",error_message1,"CBF_IDENTICAL    ");
  if (error_status & CBF_NOTFOUND      )
    sprintf(error_message,"%s %s",error_message1,"CBF_NOTFOUND     ");
  if (error_status & CBF_OVERFLOW      )
    sprintf(error_message,"%s %s",error_message1,"CBF_OVERFLOW     ");
  if (error_status & CBF_UNDEFINED     )
    sprintf(error_message,"%s %s",error_message1,"CBF_UNDEFINED    ");
  if (error_status & CBF_NOTIMPLEMENTED)
    sprintf(error_message,"%s %s",error_message1,"CBF_NOTIMPLEMENTED");
  if (error_status & CBF_NOCOMPRESSION)
    sprintf(error_message,"%s %s",error_message1,"CBF_NOCOMPRESSION");
}


%} // End of code which is not wrapped but needed to compile
◇
```

File defined by 5b, 9.

"pycbf.i" 9≡

⟨ *Constants used for compression* 3a ⟩

⟨ *Constants used for headers* 3b ⟩

⟨ *Constants used to control CIF parsing* 4a ⟩

⟨ *Constants used for encoding* 4b ⟩

⟨ *Exception handling* 5a ⟩

```
%include "cbfgenericwrappers.i"


// cbf_goniometer object

%include "cbfgoniometerwrappers.i"
```

```
%include "cbfdetectorwrappers.i"

// cbfhandle object
%include "cbfhandlewrappers.i"

◇
```

File defined by 5b, 9.

Despite the temptation to just throw everything from the c header files into the interface, a short experience suggested we are better off to pull out only the parts we want and make the calls more pythonic

The input files "CBFhandlewrappers.i", etc. are created by the make_pycbf.py script.

3.2 Exceptions

We attempt to catch the errors and pass them back to python as exceptions. This could still do with a little work to propagage back the calls causing the errors.

Currently there are two global constants defined, called error_message and error_status. These are filled out when an error occurred, converting the numerical error value into something the author can read.

There is an implicit assumption that if the library is used correctly you will not normally get exceptions. This should be addressed further in areas like file opening, proper python exceptions should be returned.

See the section on exception handling in pycbf.i, above.

Currently you get a meaningful string back. Should perhaps look into defining these as python exception classes? In any case - the SWIG exception handling is defined via the following. It could have retained the old style if(status = action) but then harder to see what to return...

4 Docstrings

The file doc/CBFlib.html is converted to a file CBFlib.txt to generate the docstrings and many of the wrappers. The conversion was done by the text-based browser, links.

This text document is then parsed by a python script called make_pycbf.py to generate the .i files which are included by the swig wrapper generator. Unfortunately this more complicated for non-python users but seemed less error prone and involved less typing for the author.

5 Wrappers

The program that does the conversion from CBFlib.txt to the SWIG input files is a python script named make_pycbf.py.

6 Building python extensions - the setup file

Based on the contents of the makefile for CBFlib we will just pull in all of the library for now. We use the distutils approach.

"py2setup_py.m4" 11a≡

```
#
#  py2setup_py.m4
#

‘# Import the things to build python binary extensions

from distutils.core import setup, Extension

# Make our extension module

e = Extension(’’‘_py2cbf’’‘,
              sources = ["pycbf_wrap.c","../src/cbf_simple.c"],
         extra_compile_args=["-g"],
         ’m4_ifelse(regexlibdir,‘NOREGEXLIBDIR’,‘library_dirs=["../solib/","../lib/"],’,‘library_dirs=["../soli
         ’m4_ifelse(regexlib,‘’,‘libraries=["cbf"],’, ‘m4_ifelse(regexlib2,‘’,‘libraries=["cbf","’regexlib‘"],
         include_dirs = ["../include","’hdf5_prefix‘/include"] )

# Build it
setup(name="_py2cbf",ext_modules=[e],)’
◇
```

7 Building and testing the resulting package

Aim to build and test in one go (so that the source and the binary match!!)

"win32.bat" 11b≡

```
nuweb pycbf
latex pycbf
nuweb pycbf
latex pycbf
dvipdfm pycbf
nuweb pycbf
C:\python24\python make_pycbf.py > TODO.txt
"C:\program files\swigwin-1.3.31\swig.exe" -python pycbf.i
C:\python24\python setup.py build --compiler=mingw32
copy build\lib.win32-2.4\_py2cbf.pyd .
REM C:\python24\python pycbf_test1.py
C:\python24\python pycbf_test2.py
C:\python24\python pycbf_test3.py
C:\python24\lib\pydoc.py -w pycbf
C:\python24\python makeflatascii.py pycbf_ascii_help.txt
◇
```

"linux.sh" 12a≡

```
nuweb pycbf
latex pycbf
nuweb pycbf
latex pycbf
dvipdfm pycbf
nuweb pycbf
lynx -dump CBFlib.html > CBFlib.txt
python make_pycbf.py
swig -python pycbf.i
python setup.py build
rm _py2cbf.so
cp build/lib.linux-i686-2.4/_py2cbf.so .
python pycbf_test1.py
python pycbf_test2.py
pydoc -w pycbf
python makeflatascii.py pycbf_ascii_help.txt
◇
```

This still gives bold in the ascii (=sucks)

"makeflatascii.py" 12b≡

```
import pydoc, pycbf, sys
f = open(sys.argv[1],"w")
pydoc.pager=lambda text: f.write(text)
pydoc.TextDoc.bold = lambda self,text : text
pydoc.help(pycbf)
◇
```

8 Debugging compiled extensions

Since it can be a bit of a pain to see where things go wrong here is a quick recipe for poking around with a debugger:

```
amber $> gdb /bliss/users//blissadm/python/bliss_python/suse82/bin/python
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-suse-linux"...
(gdb) br _PyImport_LoadDynamicModule
Breakpoint 1 at 0x80e4199: file Python/importdl.c, line 28.
```

This is how to get a breakpoint when loading the module

```
(gdb) run
Starting program: /mntdirect/_bliss/users/blissadm/python/bliss_python/suse82/bin/python
[New Thread 16384 (LWP 18191)]
Python 2.4.2 (#3, Feb 17 2006, 09:12:13)
[GCC 3.3 20030226 (prerelease) (SuSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pycbf
[Switching to Thread 16384 (LWP 18191)]

Breakpoint 1, _PyImport_LoadDynamicModule (name=0xbfffd280 "_py2cbf.so",
    pathname=0xbfffd280 "_py2cbf.so", fp=0x819e208) at Python/importdl.c:28
28              if ((m = _PyImport_FindExtension(name, pathname)) != NULL) {
```

```
(gdb) finish
Run till exit from #0  _PyImport_LoadDynamicModule (
    name=0xbfffd280 "_py2cbf.so", pathname=0xbfffd280 "_py2cbf.so", fp=0x819e208)
    at Python/importdl.c:28
load_module (name=0xbfffd710 "_py2cbf", fp=0x819e208,
    buf=0xbfffd280 "_py2cbf.so", type=3, loader=0x405b44f4)
    at Python/import.c:1678
1678                    break;
Value returned is $1 = (PyObject *) 0x405662fc
(gdb) break cbf_read_file
Breakpoint 2 at 0x407f0508: file ../src/cbf.c, line 221.
(gdb) cont
Continuing.
```

We now have a breakpoint where we wanted inside the dynamically loaded file.

```
>>> o=pycbf.cbf_handle_struct()
>>> o.read_file("../img2cif_packed.cif",pycbf.MSG_DIGEST)

Breakpoint 2, cbf_read_file (handle=0x81f7c08, stream=0x8174f58,
    headers=136281096) at ../src/cbf.c:221
221       if (!handle)
(gdb)
```

Now you can step through the c...

9 Things which are currently missing

This is the to do list. Obviously we could benefit a lot from more extensive testing and checking of the docstrings etc.

```
This output comes from make_pycbf.py which generates the wrappers
End of output from make_pycbf.py
```

10 Testing

Some test programs to see if anything appears to work. Eventually it would be good to write a proper unit test suite.

10.1 Read a file based on cif2cbf.c

This is a pretty ugly translation of the program cif2cbf.c skipping all of the writing parts. It appeared to work with the file img2cif_packed.cif which is built when you build CBFlib, hence that file is hardwired in.

"pycbf_test1.py" 14≡

```
import pycbf
object = pycbf.cbf_handle_struct() # FIXME
object.read_file("../img2cif_packed.cif",pycbf.MSG_DIGEST)
object.rewind_datablock()
print "Found",object.count_datablocks(),"blocks"
object.select_datablock(0)
print "Zeroth is named",object.datablock_name()
object.rewind_category()
categories = object.count_categories()
for i in range(categories):
    print "Category:",i,
    object.select_category(i)
    category_name = object.category_name()
    print "Name:",category_name,
    rows=object.count_rows()
    print "Rows:",rows,
    cols = object.count_columns()
    print "Cols:",cols
    loop=1
    object.rewind_column()
    while loop is not 0:
        column_name = object.column_name()
        print "column name \"",column_name,"\"",
        try:
           object.next_column()
        except:
           break
    print
    for j in range(rows):
        object.select_row(j)
        object.rewind_column()
        print "row:",j
        for k in range(cols):
            name=object.column_name()
            print "col:",name,
            object.select_column(k)
            typeofvalue=object.get_typeofvalue()
            print "type:",typeofvalue
            if typeofvalue.find("bnry") > -1:
                print "Found the binary!!",
                s=object.get_integerarray_as_string()
                print type(s)
                print dir(s)
                print len(s)
                try:
                   import numpy
                   d = numpy.frombuffer(s,numpy.uint32)
                   # Hard wired Unsigned Int32
                   print d.shape
                   print d[0:10],d[d.shape[0]/2],d[-1]
                   print d[d.shape[0]/3:d.shape[0]/3+20]
                   d=numpy.reshape(d,(2300,2300))
#                   from matplotlib import pylab
#                   pylab.imshow(d,vmin=0,vmax=1000)
#                   pylab.show()
                except ImportError:
                   print "You need to get numpy and matplotlib to see the data"
            else:
                value=object.get_value()
                print "Val:",value,i
    print
del(object)
#
```

10.2 Try to test the goniometer and detector

Had some initial difficulties but then downloaded an input cbf file which defines a goniometer and detector. The file was found in the example data which comes with CBFlib.

This test is clearly minimalistic for now - it only checks the objects for apparent existence of a single member function.

"pycbf_test2.py" 15a≡

```
import pycbf
obj = pycbf.cbf_handle_struct()
obj.read_file("../adscconverted.cbf",0)
obj.select_datablock(0)
g = obj.construct_goniometer()
print "Rotation axis is",g.get_rotation_axis()
d = obj.construct_detector(0)
print "Beam center is",d.get_beam_center()
print "Detector slow axis is", d.get_detector_axis_slow()
print "Detector fast axis is", d.get_detector_axis_fast()
print "Detector axes (fast, slow) are", d.get_detector_axes_fs()
```

◇

It appears to work - eventually. Surprising

10.3 Test cases for the generics

"pycbf_test3.py" 15b≡

```
import pycbf, unittest
class GenericTests(unittest.TestCase):

    def test_get_local_integer_byte_order(self):
        self.assertEqual( pycbf.get_local_integer_byte_order(),
                          'little_endian')

    def test_get_local_real_byte_order(self):
        self.assertEqual( pycbf.get_local_real_byte_order() ,
                          'little_endian')

    def test_get_local_real_format(self):
        self.assertEqual( pycbf.get_local_real_format(),
                          'ieee 754-1985')

    def test_compute_cell_volume(self):
        self.assertEqual( pycbf.compute_cell_volume((2.,3.,4.,90.,90.,90.)),
                           24.0)
if __name__=="__main__":
    unittest.main()
```

◇

10.4 Version of pycbf$_{t}est1withwritelogicadded$

"pycbf_test4.py" 17≡

```
# version of pycbf_test1 with write logic added
import pycbf
object = pycbf.cbf_handle_struct()
newobject = pycbf.cbf_handle_struct()
object.read_file("../img2cif_packed.cif",pycbf.MSG_DIGEST)
object.rewind_datablock()
print "Found",object.count_datablocks(),"blocks"
object.select_datablock(0)
print "Zeroth is named",object.datablock_name()
newobject.force_new_datablock(object.datablock_name());
object.rewind_category()
categories = object.count_categories()
for i in range(categories):
    print "Category:",i,
    object.select_category(i)
    category_name = object.category_name()
    print "Name:",category_name,
    newobject.new_category(category_name)
    rows=object.count_rows()
    print "Rows:",rows,
    cols = object.count_columns()
    print "Cols:",cols
    loop=1
    object.rewind_column()
    while loop is not 0:
        column_name = object.column_name()
        print "column name \"",column_name,"\"",
        newobject.new_column(column_name)
        try:
            object.next_column()
        except:
            break
    print
    for j in range(rows):
        object.select_row(j)
        newobject.new_row()
        object.rewind_column()
        print "row:",j
        for k in range(cols):
            name=object.column_name()
            print "col:",name,
            object.select_column(k)
            newobject.select_column(k)
            typeofvalue=object.get_typeofvalue()
            print "type:",typeofvalue
            if typeofvalue.find("bnry") > -1:
                print "Found the binary!!",
                s=object.get_integerarray_as_string()
                print type(s)
                print dir(s)
                print len(s)
                (compression, binaryid, elsize, elsigned, \
                    elunsigned, elements, minelement, maxelement, \
                    byteorder,dimfast,dimmid,dimslow,padding) = \
                    object.get_integerarrayparameters_wdims_fs()
                if dimfast==0:
                    dimfast = 1
                if dimmid==0:
                    dimmid = 1
                if dimslow == 0:
                    dimslow = 1
                print "compression: ",compression
```

10.5 Processing of XFEL axes

"pycbf_testfelaxes.py" 18≡

```
import pycbf, sys
from decimal import Decimal, ROUND_HALF_UP

image_file = sys.argv[1]

cbf = pycbf.cbf_handle_struct()
cbf.read_widefile(image_file, pycbf.MSG_DIGEST)

for element in range(64):
    d = cbf.construct_detector(element)
    print "element:", element

    v00 = d.get_pixel_coordinates(0, 0)
    v01 = d.get_pixel_coordinates(0, 1)
    v10 = d.get_pixel_coordinates(1, 0)
    v11 = d.get_pixel_coordinates(1, 1)
    prec = Decimal('1.000000000')

    print '(0, 0) v00 [ %.9f %.9f %.9f ]' %(round(v00[0],9), round(v00[1],9), round(v00[2],9))
    print '(0, 1) v01 [ %.9g %.9g %.9g ]' %(round(v01[0],9), round(v01[1],9), round(v01[2],9))
    print '(1, 0) v10 [ %.9g %.9g %.9g ]' %(round(v10[0],9), round(v10[1],9), round(v10[2],9))
    print '(1, 1) v11 [ %.9g %.9g %.9g ]' %(round(v11[0],9), round(v11[1],9), round(v11[2],9))

    print "surface axes:",  d.get_detector_surface_axes(0), d.get_detector_surface_axes(1)

    print d.get_detector_surface_axes(0), "has", cbf.count_axis_ancestors(d.get_detector_surface_axes(0)), "anc
    print d.get_detector_surface_axes(1), "has", cbf.count_axis_ancestors(d.get_detector_surface_axes(1)), "anc

    cur_axis = d.get_detector_surface_axes(0)
    count = cbf.count_axis_ancestors(cur_axis)

    for index in range(count):
        print "axis", cur_axis, "index: ", index
        print "    equipment", cbf.get_axis_equipment(cur_axis)
        print "    depends_on", cbf.get_axis_depends_on(cur_axis)
        print "    equipment_component", cbf.get_axis_equipment_component(cur_axis)
        vector = cbf.get_axis_vector(cur_axis)
        print "    vector [ %.8g %.8g %.8g ]" % (round(vector[0],7), round(vector[1],7), round(vector[2],7))
        offset = cbf.get_axis_offset(cur_axis)
        print "    offset [ %.8g %.8g %.8g ]" % (round(offset[0],7), round(offset[1],7), round(offset[2],7))
        print "    rotation", cbf.get_axis_rotation(cur_axis)
        print "    rotation_axis", cbf.get_axis_rotation_axis(cur_axis)
        cur_axis = cbf.get_axis_depends_on(cur_axis)
```

◇

11 Worked example 1 : xmas beamline + mar ccd detector at the ESRF

Now for the interesting part. We will attempt to actually use pycbf for a real dataprocessing task. Crazy you might think.

The idea is the following - we want to take the header information from some mar ccd files (and eventually also the user or the spec control system) and pass this information into cif headers which can be read by fit2d (etc).

11.1 Reading marccd headers

Some relatively ugly code which parses a c header and then tries to interpret the mar ccd header format.

FIXME : byteswapping and ends???

"xmas/readmarheader.py" 20≡

```
#!/usr/bin/env python
import struct

# Convert mar c header file types to python struct module types
mar_c_to_python_struct = {
    "INT32"  : "i",
    "UINT32" : "I",
    "char"   : "c",
    "UINT16" : "H"
    }

# Sizes (bytes) of mar c header objects
mar_c_sizes = {
    "INT32"  : 4,
    "UINT32" : 4,
    "char"   : 1,
    "UINT16" : 2
    }

# This was worked out by trial and error from a trial image I think
MAXIMAGES=9



def make_format(cdefinition):
    """
    Reads the header definition in c and makes the format
    string to pass to struct.unpack
    """
    lines = cdefinition.split("\n")
    fmt = ""
    names = []
    expected = 0
    for line in lines:
        if line.find(";")==-1:
            continue
        decl  = line.split(";")[0].lstrip().rstrip()
        try:
            [type, name] = decl.split()
        except:
            #print "skipping:",line
            continue
        #        print "type:",type,"  name:",name

        if name.find("[")>-1:
            # repeated ... times
            try:
                num = name.split("[")[1].split("]")[0]
                num = num.replace("MAXIMAGES",str(MAXIMAGES))
                num = num.replace("sizeof(INT32)","4")
                times = eval(num)
            except:
                print "Please decode",decl
                raise
        else:
            times=1
        try:
            fmt   += mar_c_to_python_struct[type]*times
            names += [name]*times
            expected += mar_c_sizes[type]*times
        except:
            #print "skipping",line
            continue
        #print "%4d %4d"%(mar_c_sizes[type]*times,expected),name,":",times,line
```

11.2 Writing out cif files for fit2d/xmas

A script which is supposed to pick up some header information from the mar images, some more infomation from the user and the create cif files.

This relies on a "template" cif file to get it started (avoids me programming everything).

"xmas/xmasheaders.py" 22≡

```
#!/usr/bin/env python


import pycbf

# Some cbf helper functions - obj would be a cbf_handle_struct object

def writewavelength(obj,wavelength):
    obj.set_wavelength(float(wavelength))

def writecellpar(obj,cifname,value):
    obj.find_category("cell")
    obj.find_column(cifname)
    obj.set_value(value)

def writecell(obj,cell):
    """
    call with cell = (a,b,c,alpha,beta,gamma)
    """
    obj.find_category("cell")
    obj.find_column("length_a")
    obj.set_value(str(cell[0]))
    obj.find_column("length_b")
    obj.set_value(str(cell[1]))
    obj.find_column("length_c")
    obj.set_value(str(cell[2]))
    obj.find_column("angle_alpha")
    obj.set_value(str(cell[3]))
    obj.find_column("angle_beta")
    obj.set_value(str(cell[4]))
    obj.find_column("angle_gamma")
    obj.set_value(str(cell[5]))

def writeUB(obj,ub):
    """
    call with ub that can be indexed ub[i][j]
    """
    obj.find_category("diffrn_orient_matrix")
    for i in (1,2,3):
        for j in (1,2,3):
            obj.find_column("UB[%d][%d]"%(i,j))
            obj.set_value(str(ub[i-1][j-1]))

def writedistance(obj,distance):
    obj.set_axis_setting("DETECTOR_Z",float(distance),0.)


def writebeam_x_mm(obj,cen):
    obj.set_axis_setting("DETECTOR_X",float(cen),0.)

def writebeam_y_mm(obj,cen):
    obj.set_axis_setting("DETECTOR_Y",float(cen),0.)

def writeSPECcmd(obj,s):
    obj.find_category("diffrn_measurement")
    obj.find_column("details")
    obj.set_value(s)

def writeSPECscan(obj,s):
    obj.find_category("diffrn_scan")
    obj.find_column("id")
    obj.set_value("SCAN%s"%(s))
    obj.find_category("diffrn_scan_axis")
    obj.find_column("scan_id")
```

11.3 A template cif file for the xmas beamline

This was sort of copied and modified from an example file. It has NOT been checked. Hopefully the four circle geometry at least vaguely matches what is at the beamline.

"xmas/xmas_cif_template.cif" 24≡

```
###CBF: VERSION 0.6
# CBF file written by cbflib v0.6



data_image_1



loop_
_diffrn.id
_diffrn.crystal_id
 DS1 DIFFRN_CRYSTAL_ID

loop_
_cell.length_a                    5.959(1)
_cell.length_b                    14.956(1)
_cell.length_c                    19.737(3)
_cell.angle_alpha                 90
_cell.angle_beta                  90
_cell.angle_gamma                 90


loop_
_diffrn_orient_matrix.id 'DS1'
_diffrn_orient_matrix.type
; reciprocal axis matrix, multiplies hkl vector to generate
  diffractometer xyz vector and diffractometer angles
;
_diffrn_orient_matrix.UB[1][1]           0.11
_diffrn_orient_matrix.UB[1][2]           0.12
_diffrn_orient_matrix.UB[1][3]           0.13
_diffrn_orient_matrix.UB[2][1]           0.21
_diffrn_orient_matrix.UB[2][2]           0.22
_diffrn_orient_matrix.UB[2][3]           0.23
_diffrn_orient_matrix.UB[3][1]           0.31
_diffrn_orient_matrix.UB[3][2]           0.32
_diffrn_orient_matrix.UB[3][3]           0.33




loop_
_diffrn_source.diffrn_id
_diffrn_source.source
_diffrn_source.current
_diffrn_source.type
 DS1 synchrotron 200.0 'XMAS beamline bm28 ESRF'

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
 DS1 WAVELENGTH1 x-ray 'Si 111' 0.8 0.0 0.08 0.01 0.00 '0.20 mm x 0.20 mm'

loop_
```