International Tables for Crystallography (2006). Vol. G, Section 5.4.3, pp. 526–527.

5.4. *CIFtbx*: Fortran tools for manipulating CIFs

BY H. J. BERNSTEIN AND S. R. HALL

5.4.1. Introduction

CIFtbx is a function library for programmers developing CIF applications. It is written in Fortran and is intended for use with Fortran programs. The first version was released in 1993 (Hall, 1993*b*) and was extended (Hall & Bernstein, 1996) to accommodate subsequent CIF applications and DDL changes. The *CIFtbx* library is for novice and expert programmers of CIF applications. It has been used to develop CIF manipulation programs such as *CYCLOPS* (Bernstein & Hall, 1998), *CIFIO* (Hall, 1993*a*), *cif2cif* (Bernstein, 1997), *pdb2cif* (Bernstein *et al.*, 1998) and *cif2pdb* (Bernstein & Bernstein, 1996). Programmers writing in C, C++ and mixed Fortran–C should consider alternative approaches, as discussed in Chapter 5.1 or in the work on *CCP4* (Keller, 1996).

The description of library functions below assumes familiarity with the STAR, CIF and DDL syntax described in Part 2. A complete *Primer and reference manual* for *CIFtbx* is provided on the CD-ROM accompanying this volume.

Fortran is a very general and powerful language, and many compilers allow programming in a wide variety of styles. However, there is a traditional Fortran programming style that ensures portability to a wide variety of platforms. *CIFtbx* conforms to this style and has been ported to many platforms. The internals of *CIFtbx* and the style chosen are discussed at the end of this chapter and in more detail in the *Primer*.

5.4.2. An overview of the library

The *CIFtbx* library is made up of functions, subroutines and variables that can be added to application programs as 'commands' to read and write CIF data. They may also be used to automatically validate incoming and outgoing CIF data. The self-checking aspects of some functions ensure that data are syntactically correct and, when used with DDL dictionaries, that individual items conform to their formal definitions.

The *CIFtbx* commands are invoked in user software as standard Fortran function or subroutine calls. For example, to open the dictionary file 'core.dic' one uses the *logical* function `dict_` as follows.

```
FN = dict_('core.dic','valid')
```

The argument `'core.dic'` is the local file identifier for the relevant dictionary. The argument `'valid'` signals that checking should be done against the data definitions in this dictionary. The local *logical* variable `FN` is returned as `.true.` if `dict_` opens the file core.dic correctly; otherwise the function is returned as `.false.`.

Some *CIFtbx* commands are issued as subroutine calls. For example, to clear the internal data tables the programmer inserts the command

```
call purge_
```

The arguments in *CIFtbx* commands have been kept to a minimum. Most of the parameter setting is handled automatically by reading and setting variables held in common blocks supplied as the file ciftbx.cmn. The type declarations for all the commands are also provided in the file ciftbx.cmn, and the programmer must '`include`' this file in each application program, function or subroutine invoking *CIFtbx* commands.

The flexibility of the CIF syntax can present some challenges to an author of applications reading or writing CIF data. This is because the information in a CIF may be in any order, have data names as either upper or lower case, and have an arbitrary spacing between data items. For example, one may extract the cell parameters from the front of a CIF and place them at the end, change all the data names from lower case to upper case, and introduce a blank line between each data name and its value, and yet the data (value) content of the output CIF will be identical to that input. *CIFtbx* provides the application writer with the tools to handle such presentation details seamlessly without altering the basic information content.

Most importantly, *CIFtbx* allows applications to be 'object-oriented', in that data items are simply requested by name without prior knowledge of the file structure. It also allows for more advanced data processing in which data items are parsed sequentially, and typed and validated *via* the dictionary. This enables items to be read independently of the names, and the data typing is automatically determined and returned. In this way, where needed, applications can go beyond the position-independent context of a CIF.

The main purpose of *CIFtbx* is to manipulate CIF data. However, there is much in common between CIF and the Extensible Markup Language XML (Bray *et al.*, 1998), and facilities have been added to *CIFtbx* to facilitate writing output in XML as well as CIF format.

CIFtbx provides four basic kinds of facilities for programmers:
(i) commands to *initialize* later handling;
(ii) commands to *read* CIF data;
(iii) commands to *write* CIF data;
(iv) variables for *monitor* and *control* signals.
These commands are described in detail below.

5.4.3. Initialization commands

Initialization commands are applied at the start of a program to set global conditions for processing CIF data. There are only two commands of this type.

```
logical function init_
  (devcif, devout, devdir, deverr)
  integer devcif, devout, devdir, deverr
logical function dict_ (fname, checks)
  character fname*(*), checks*(*)
```

`init_` is an optional command that specifies the device number assignments for the input CIF *`devcif`*, the output CIF *`devout`*, an internal scratch file *`devdir`* and the file containing error messages *`deverr`*. The internal scratch file *`devdir`* is used to hold a copy of

Affiliations: HERBERT J. BERNSTEIN, Department of Mathematics and Computer Science, Kramer Science Center, Dowling College, Idle Hour Blvd, Oakdale, NY 11769, USA; SYDNEY R. HALL, School of Biomedical and Chemical Sciences, University of Western Australia, Crawley, Perth, WA 6009, Australia.

the input CIF as a direct-access file (*i.e.* for random access to parts of the CIF). `init_` is a *logical* function that is always returned with a value of `.true.`. The default device numbers for these files are 1, 2, 3 and 6.

`dict_` is an optional command for opening a dictionary *`fname`* and initiating various optional data checks, *`checks`*. The choices of checks to perform are given by a string of blank-separated five-character 'check codes', such as `valid` or `dtype`, which turn on checking for the validity of tags or types of values, respectively. `dict_` is a *logical* function which is returned as `.true.` if the named dictionary was opened and if the check codes are recognizable.

5.4.4. Read commands

These commands are used to read data from an existing CIF. Since CIF data are order-independent, most applications would work from a known list of data names (tags) and extract the desired values from the CIF in the order specified. However, some applications need to browse a CIF in the order of presentation. In *CIFtbx*, a blank name has the meaning of the next name in the file.

```
logical function ocif_ (fname)
  character fname*(*)
logical function data_ (name)
  character name*(*)
logical function bkmrk_ (mark)
  integer mark
logical function find_ (name, type, strg)
  character name*(*), type*(*), strg*(*)
logical function test_ (name)
  character name*(*)
logical function name_ (name)
  character name*(*)
logical function numb_ (name, numb, sdev)
  character name*(*)
  real numb, sdev
logical function numd_ (name, numb, sdev)
  character name*(*)
  double precision numb, sdev
logical function char_ (name, strg)
  character name*(*), strg*(*)
logical function cmnt_ (strg)
  character strg*(*)
subroutine purge_
```

`ocif_` requests the named CIF *`fname`* to be opened. The *logical* function is returned as `.true.` if the CIF can be opened.

`data_` specifies the data block *`name`* containing the data to be read from the CIF. The *logical* function is returned as `.true.` if the data block is found.

`bkmrk_` is a bookmark function that saves or restores the current position in the CIF so that data can be accessed nonsequentially if need be. The *logical* function is returned as `.true.` if there is space to store the current position or if the restored bookmark number is valid.

`find_` finds the requested item in the current data block. The *logical* function is returned as `.true.` if the item is found.

`test_` provides the data attributes of a data item in the current data block. The *logical* function is returned as `.true.` if the item is found. The data attributes are returned in the common-block variables `list_`, `type_`, `dictype_`, `diccat_` and `dicname_`.

`name_` identifies the next data name in the current data block. The *logical* function is returned as `.true.` if another data name exists in the data block and `.false.` if the end of the data block is reached. The name is returned in the function argument, *`name`*.

`numb_` returns the number *`numb`* and its standard uncertainty *`sdev`* (if appended) of a named data item *`name`*. The *logical* function is returned as `.true.` if the item is present and is a number. If the item is either absent or cannot be recognized as a valid number, the function is returned as `.false.` and the original numeric argument values are not changed.

`numd_` returns the number *`numb`* and its standard uncertainty *`sdev`* (if appended) as double-precision variables of a named data item *`name`*. The *logical* function is returned as `.true.` if the item is present and is a number. If the item is either absent or cannot be recognized as a valid number, the function is returned as `.false.` and the original numeric argument values are not changed.

`char_` returns character or text strings, *`strg`*, of the named data item *`name`*. The *logical* function is returned as `.true.` if the item is present. If text lines are being read, this function is called repeatedly until the *logical variable* `text_` is `.false.`.

`cmnt_` returns the next comment, *`strg`*, in the current data block. The *logical* function is returned as `.true.` if a comment is present. The initial comment character '`#`' is not included in the returned string and a completely blank line is treated as a comment.

`purge_` closes all attached data files and clears all tables and pointers. This is a subroutine call.

5.4.5. Write commands

The following commands are available for writing data to a new CIF.

```
logical function pfile_ (fname)
  character fname*(*)
logical function pdata_ (name)
  character name*(*)
logical function ploop_ (name)
  character name*(*)
logical function pnumb_ (name, numb, sdev)
  character name*(*)
  real numb, sdev
logical function pnumd_ (name, numb, sdev)
  character name*(*)
  double precision numb, sdev
logical function pchar_ (name, string)
  character name*(*), string*(*)
logical function pcmnt_ (string)
  character string*(*)
logical function ptext_ (name, string)
  character name*(*), string*(*)
logical function prefx_ (strg, lstrg)
  character strg*(*)
  integer lstrg
subroutine close_
```

`pfile_` creates a new file with the specified file name *`fname`*. The *logical* function is returned as `.true.` if the file is opened. The value will be `.false.` if the file already exists.

`pdata_` puts the string **`data_`*`name`*** from the argument *`name`* into the output CIF. The *logical* function is returned as `.true.` if the block is created. The value will be `.false.` if the block name already exists. This command inserts the string **`save_`*`name`*** instead of the data-block name if the variable `saveo_` is set to `.true.`. If the prior block was a save frame, the necessary terminal **`save_`** is written for that block before the new block is started.

`ploop_` puts the specified data name *`name`* into the output CIF. On the first invocation of this command for a given loop, a **`loop_`** string is placed before the data name. The *logical* function is returned as `.true.` if the name passes any requested dictionary validation checks. Once a series of data names for a **`loop_`** header has been declared by calls to this function, all calls to `pchar_`, `ptext_`, `pnumb_` or `pnumd_` for the associated data values must be made with *blank* data names or the **`loop_`** will be terminated. (At the very least, the first character of these data names must be blank.)

`pchar_` puts the specified data name *`name`* and *character string* *`string`* into the output CIF. If the data name is blank, only the