International Tables for Crystallography (2006). Vol. G, Chapter 5.4, pp. 526–538.

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

character string is put. The *logical* function is returned as `.true.` if the data name passes any requested dictionary validation checks.

`pnumb_` puts the specified data name *`name`*, single-precision number *`numb`* and an appended standard uncertainty *`sdev`* into the output CIF. The *logical* function is returned as `.true.` if the data name passes any requested dictionary validation checks.

`pnumd_` puts the specified data name *`name`*, double-precision number *`numb`* and an appended standard uncertainty *`sdev`* into the output CIF. The *logical* function is returned as `.true.` if the data name passes any requested dictionary validation checks.

`ptext_` puts the specified data name *`name`* and text string *`string`* into the output CIF. The data name will only be inserted on the first invocation of a sequence. The *logical* function is returned as `.true.` if the data name passes any requested dictionary validation checks. This command must be invoked repeatedly until the text is finished. The terminal semicolon character '`;`' is placed in the output CIF when the next call to `pchar_`, `pnumb_` or `pnumd_` is made, or if a call is made to `ptext_` for a different data name.

`pcmnt_` puts the specified comment string *`string`* into the output CIF. The *logical* function is always returned as `.true.`. The comment character '`#`' should not be included in the string. A blank comment is presented as a blank line without the leading '`#`'. The string `char(0)//char(0)` can be used to produce an empty comment with the leading '`#`'.

`prefx_` prefixes the specified string *`strg`* of length *`lstrg`* to subsequent lines of the output CIF. The total line length is still limited to the value given by the variable `line_` (the default is 80 characters). This function is useful when embedding a CIF into another text document, such as a PDB REMARK. The *logical* function is always returned as `.true.`.

`close_` closes the output CIF only. This command *must* be used if `pfile_` is used. This a subroutine call.

5.4.6. Variables

The *CIFtbx* library also contains a large number of variables declared in the common blocks in the file ciftbx.cmn that provide signals to the programmer on various aspects of the data reading and writing processes. These variables are described below in four broad categories, as shown in Table 5.4.6.1: general monitor variables, general control variables, input monitor variables and output control variables.

Note that for all but special applications only the basic variables `list_`, `loop_`, `strg_`, `text_` and `type_` are usually used. These variables supplement the argument lists of the various commands, providing essential status information.

5.4.6.1. General monitor variables

These variables are returned by *CIFtbx* and provide information about the general status of processing.

`file_`: *character string* containing the file name of the current input file.

`longf_`: *integer variable* containing the length of the file name in *`file`*.

`precn_`: *integer variable* containing the line number (starting from 1) of the last line written to the output CIF.

`recn_`: *integer variable* containing the line number (starting from 1) of the last line read from the input CIF.

`tbxver_`: *character*32 variable* that is the *CIFtbx* version and date in the form *`'CIFtbx version N.N.N DD MMM YYYY'`* (some older versions of *CIFtbx* use a two-digit year and have a comma after the version number).

Table 5.4.6.1. *CIFtbx variables*

General monitor	General control	Input monitor	Output control
	`alias_`		`aliaso_`
			`align_`
	`append_`		
		`bloc_`	
		`decp_`	`pdecp_`
		`diccat_`	
		`dicname_`	
		`dictype_`	
		`dicver_`	
			`esdlim_`
`file_`			
		`glob_`	`globo_`
	`line_`		
		`list_`	
		`long_`	
`longf_`			
		`loop_`	
		`lzero_`	`plzero_`
	`nblank_`		`nblanko_`
		`posdec_`	`pposdec_`
		`posend_`	`pposend_`
		`posnam_`	`pposnam_`
		`posval_`	`pposval_`
`precn_`			
		`quote_`	`pquote_`
	`recbeg_`		
	`recend_`		
`recn_`			
		`save_`	`saveo_`
		`strg_`	
			`tabl_`
	`tabx_`		`ptabx_`
		`tagname_`	
`tbxver_`			
		`text_`	
		`type_`	
			`xmlout_`
			`xmlong_`

5.4.6.2. General control variables

These variables control *CIFtbx* commands. The user may accept the default values or may store new values into these variables to change the behaviour of the commands.

`alias_`: *logical variable* to control the use of data-name aliases for input items. If set to `.true.`, aliases from the input dictionary may be used (see Section 5.4.7). The default is `.true.`.

`append_`: *logical variable* to control reuse of the direct-access file. If set to `.true.`, it will cause each call to `ocif_` to append the information found to the current CIF. The default is `.false.`.

`line_`: *integer variable* to set the input/output line limit for processing a CIF. The default value is 80 characters. This limit counts the visible printable characters of the line, not the system-dependent line terminators.

`nblank_`: *logical variable* to control the treatment of input blank strings. If set to `.true.`, `char_` or `test_`, it will return the type as 'null' rather than 'char' when encountering a quoted blank.

`recbeg_`: *integer variable* to give the record number of the first record to be used. May be changed by the user to restrict access to a CIF.

`recend_`: *integer variable* to give the record number of the last record to be used. May be changed by the user to restrict access to a CIF.

`tabx_`: *logical variable* is set to `.true.` for tab stops to be expanded to blanks during the reading of a CIF. The default is `.true.`.

5.4.6.3. Input monitor variables

These variables are returned by *CIFtbx* tools and are used to decide on subsequent actions in the program. The lengths of the character strings that hold data names and block names are controlled by the parameter NUMCHAR in the common-block declarations.

bloc_: *character string* containing the current data-block name.

decp_: *logical variable* is .true. if a decimal point is present in the input numeric value.

diccat_: *character string* containing the category name specified in the attached dictionaries.

dicname_: *character string* containing the root alias data name (see Section 5.4.7) specified in the attached dictionaries or, after a call to dict_, the name of the dictionary.

dictype_: *character string* containing the data-type code specified in the attached dictionaries. These types may be more specific (*e.g.* 'float' or 'int') than the types given by the variable type_ (*e.g.* 'numb').

dicver_: *character string* containing the version of a dictionary after a call to dict_.

glob_: *logical variable* is .true. if the current data block is a global block. The application is responsible for managing the relationship of global data to other data blocks.

list_: *integer variable* containing the sequence number of the current looped list. This value may be used by the application to identify variables that are in different lists or that are not in a list (a zero value).

long_: *integer variable* containing the length of the data string in strg_.

loop_: *logical variable* is .true. if another loop packet is present in the current looped list.

lzero_: *logical variable* is .true. if the input numeric value is of the form [*sign*]0.*nnnn* rather than [*sign*].*nnnn*.

posdec_: *integer variable* containing the column number (position along the line, counting from 1 at the left) of the decimal point for the last number read.

posend_: *integer variable* containing the column number (position along the line, counting from 1 at the left) of the last character for the last string or number read.

posnam_: *integer variable* containing the starting column (position along the line, counting from 1 at the left) of the last name or comment read.

posval_: *integer variable* containing the starting column (position along the line, counting from 1 at the left) of the last data value read.

quote_: *character variable* giving the quotation symbol found delimiting the last string read.

save_: *logical variable* is .true. if the current data block is a save frame, otherwise .false..

strg_: *character variable* containing the data name or string representing the data value last retrieved.

tagname_: *character variable* containing the data name of the current data item as it was found in the CIF. May differ from dicname_ because of aliasing.

text_: *logical variable* is .true. if another text line is present in the current input text block.

type_: *character variable* containing the data-type code of the current input data item. This will be one of the four-character strings 'null' (for missing data, the period or the question mark), 'numb' (for numeric data), 'char' (for most character data) or 'text' (for semicolon-delimited multi-line character data). For most purposes the type 'text' is a subtype of the type 'char', not a distinct data type. *CIFtbx* permits multi-line text fields to be used whenever character strings are expected.

5.4.6.4. Output control variables

These variables are specified to control the processing by *CIFtbx* commands that write CIFs.

aliaso_: *logical variable* to control the use of data-name aliases for output items. If set to .true., preferred synonyms from the input dictionary may be output (see Section 5.4.7). The default is .false..

align_: *logical variable* to control the column alignment of data values in **loop_** lists output to a CIF. The default is .true..

esdlim_: *integer variable* to set the upper limit of appended standard uncertainty (e.s.d.) integers output by **pnumb_**. The default value is 19, which limits standard uncertainties to the range 2–19.

globo_: *logical variable* which if set to .true. will cause the output data block from pdata_ to be written as a global block.

nblanko_: *logical variable* controls the treatment of output blank strings. If set to .true., output quoted blank strings will be converted to an unquoted period (*i.e.* to a data item of type null). Recall that *CIFtbx* treats an unquoted period or question mark as being of type null.

pdecp_: *logical variable* controls the treatment of output decimal numbers. If set to .true., a decimal point will be inserted into numbers output by pnumb_ or pnumbd_. If set to .false., a decimal point will be output only when needed. The default is .false..

plzero_: *logical variable* controls the treatment of leading zeros in output decimal numbers. If set to .true., a zero will be inserted before a leading decimal point. The default is .false..

pposdec_: *integer variable* to set the column number (position along the line, counting from 1 at the left) of the decimal point for the next number to be output.

pposend_: *integer variable* to set the position of the ending column for the next number or *character string* to be output. Used to pad with zeros or blanks.

pposnam_: *integer variable* to set the starting column of the next name or comment to be output.

pposval_: *integer variable* to set the position of the starting column of the next data value to be output.

pquote_: *character variable* containing the quotation symbol to be used for the next string written.

saveo_: *logical variable* is set to .true. for pdata_ to output a save frame, otherwise a data block is output.

ptabx_: *logical variable* is set to .true. for tab stops to be expanded to blanks during the creation of a CIF. The default is .true..

tabl_: *logical variable* is set to .true. for tab stops to be used in the alignment of output data. The default is .true..

xmlout_: *logical variable* is set to .true. to change the output style to XML conventions. Note that this is not a CML (Murray-Rust & Rzepa, 1999) output, but a literal translation from the input CIF. The default is .false..

xmlong_: *logical variable* is set to .true. to change the style of XML output if xmlout_ is .true.. When .true. (the default), XML tag names are the full CIF tag names with the leading underscore, _, removed. When .false., an attempt is made to strip the leading category name as well.

5.4.7. Name aliases

CIF dictionaries written in DDL2 permit data names to be aliased or equivalenced to other data names. This serves two purposes. First, it allows for the different data-name structures used in DDL1

```
    read(8,'(a)',end=400) name
       f1 = test_(name)
       write(6,'(2(3x,a32)') name,dicname_
       name=dicname_
       f1 = test_(name)
       write(6,'(2(3x,a32)') name,tagname_
```

Fig. 5.4.7.1. Example application accessing aliased data names.

and DDL2 dictionaries, and, second, it links equivalent data names within the DDL2 dictionary. Aliasing also allows the use of synonyms appropriate to the application.

CIFtbx is capable of handling aliased data names transparently so that both the input CIF and the application software can use any of the equivalent aliased names. In addition, an output CIF may be written with the data names specified in the *CIFtbx* functions or with names that have been automatically converted to preferred dictionary names. If more than one dictionary is loaded, the first aliases have priority. We call the preferred dictionary name the 'root alias'.

The default behaviour of *CIFtbx* is to accept all combinations of aliases and to produce output CIFs with the exact names specified in the user calls. The interpretation of aliased data names is modified by setting the *logical variable*s `alias_` and `aliaso_`. When `alias_` is set to `.false.`, the automatic recognition and translation of aliases stops. When `aliaso_` is set to `.true.`, the automatic conversion of user-supplied names to dictionary-preferred alias names in writing data to output CIFs is enabled. The preferred alias name is stored in the variable `dicname_` following any invocation of a getting function, such as `numb_` or `test_`. If `alias_` is set to `.false.`, `dicname_` will correspond to the called name. The variable `tagname_` is always set to the actual name used in an input CIF.

For example, the data name **`_atom_site_anisotrop.u[1][1]`** in the DDL2 mmCIF dictionary is aliased to the data name **`_atom_site_aniso_U_11`** in the DDL1 core CIF dictionary. In the example application of Fig. 5.4.7.1, showing the *CIFtbx* function `test_` run with both the mmCIF and core dictionaries loaded, the specified data name **`_atom_site_aniso_U_11`** is used to inquire as to the names used in an input CIF.

The execution of this code results in the following printout.

```
_atom_site_aniso_U_11
          _atom_site_anisotrop.u[1][1]
_atom_site_anisotrop.u[1][1]
          _atom_site_aniso_U_11
```

5.4.8. Implementation of the tools

Implementation of the *CIFtbx* tools is straightforward. The supplied source files in all versions are: ciftbx.f, ciftbx.sys (used in ciftbx.f) and ciftbx.cmn (used in local applications). More recent versions of *CIFtbx* (version 2.4 and later) require certain additional source files: ciftbx.cmf, ciftbx.cmv, hash_funcs.f and clearfp.f (or clearfp_sun.f).

The common file ciftbx.cmn must be '`include`'d into any local routines that use *CIFtbx* tools. The library in ciftbx.f may be invoked by either (i) compiling and linking the resulting object file as an object library, or with explicit references in the application linking sequence (versions 2.4 and later require hash_funcs.o as an additional object file); or (ii) including ciftbx.f in the local application and compiling and linking it with the local program.

Approach (i) is more efficient, but for some applications approach (ii) may be simpler.

5.4.9. How to read CIF data

The *CIFtbx* approach to reading CIF data is illustrated using a simple example program *CIF_IN* (Fig. 5.4.9.1), which reads the file test.cif (Fig. 5.4.9.2) and tests the input data items against the dictionary file cif_core.dic. The resulting output is shown in Fig. 5.4.9.3.

The program *CIF_IN* may be divided into the following steps, each tagged with the relevant reference letter in the comment records of the listing shown in Fig. 5.4.9.1.

A: Define the local variables. The *CIFtbx* variables are added with the line `include 'ciftbx.cmn'`.

B: Assign device numbers to the files using the command `init_`. The device number 1 refers to the input CIF, 3 to the scratch file and 6 (stdout) to the error-message files. The device number 2 refers to an output CIF, if we were to choose to write one.

C: Open a specific dictionary file named cif_core.dic with the command `dict_`. The code `valid` signals that the input data items are to be validated against the dictionary. In this application, `dict_` is invoked in an `IF` statement that tests whether the command is successful.

D: Open the CIF test.cif with the command `ocif_` and test that the file is opened.

E: Invoke the `data_` command, containing a blank block code, to 'open' the next data block. The block name encountered is placed in the variable `bloc_`, which in this application is printed.

F: Read the cell-length values and their standard uncertainties with the `numb_` command, and print these out. Test whether all of the requested data items are found.

G: The `char_` function is used to read a single *character string*.

H: The `name_` function is used to get the data name of the next data item encountered.

I: This sequence illustrates how text lines are read. The `char_` function is used to read each line and the `text_` variable is tested to see whether another text line exists in this data item.

J: This sequence illustrates how a looped list of items is read. Individual items are read using `char_` or `numb_` functions and the existence of another packet of items is tested with the variable `loop_`.

The resulting printout is shown in Fig. 5.4.9.3. In this figure, note the following:

(i) The first six lines of the printout are output by *CIFtbx* routines, not by the program *CIF_IN*. They occur when the `data_` command is executed and data items in the block **`mumbo_jumbo`** are read from the CIF and checked against the dictionary file. Note that this is when the *CIFtbx* routines store the pointers and attributes of all items in the data block. All subsequent commands use these pointers to access the data.

(ii) The '**`####`**' string in front of **`_cell_length_a`** in the input CIF makes this line a comment and makes it inaccessible to *CIF_IN*.

(iii) Data items may be read from a CIF in any order but looped items must normally be in the same list. If one needs to access looped items in different lists simultaneously, the `bkmrk_` command is used to preserve *CIFtbx* loop pointers.

5.4.10. How to write a CIF

Writing a CIF usually is simpler than reading an existing one. An example of a CIF-writing program is shown in Fig. 5.4.10.1. This example is intentionally trivial. The created CIF test.new is shown in Fig. 5.4.10.2. Note that command `dict_` causes all output items to be checked against the dictionary cif_core.dic. Unknown names are flagged in the output CIF with the comment

```
                        PROGRAM  CIF_IN
C
C....A.. Define the data variables
         include       'ciftbx.cmn'
         logical       f1,f2,f3
         character*32  name
         character*80  line
         real          cela,celb,celc,siga,sigb,sigc
         real          x,y,z,u,numb,sdev
         data          cela,celb,celc,siga,sigb,sigc /6*0.0/

C....B.. Assign the CIFtbx files
         f1 = init_( 1, 2, 3, 6 )

C....C.. Request dictionary validation check
         if(dict_('cif_core.dic','valid'))  goto 100
         write(6,'(/a/)') ' Requested Core dictionary not present'

C....D.. Open the CIF to be accessed
100      name='test.cif'
         if(ocif_(name))      goto 120
         write(6,'(a///)')  ' >>>>>>>>> CIF cannot be opened'
         stop
C....E.. Assign the data block to be accessed
120      if(.not.data_(' '))       goto 200
         write(6,'(/a,a/)') ' Access items in data block  ',bloc_

C....F.. Extract some cell dimensions; test all is OK
         f1 = numb_('_cell_length_a', cela, siga)
         f2 = numb_('_cell_length_b', celb, sigb)
         f3 = numb_('_cell_length_c', celc, sigc)
         if(.not.(f1.and.f2.and.f3)) write(6,'(a)') ' Cell lengths missing!'
         write(6,'(a,6f10.4)') ' Cell ',cela,celb,celc,siga,sigb,sigc

C....G.. Extract space group notation (expected char string)
         f1 = char_('_symmetry_cell_setting', name)
         write(6,'(a,a/)') ' Cell setting   ',name(1:long_)

C....H.. Get the next name in the CIF and print it out
         f1 = name_(name)
         write(6,'(a,a/)') ' Next data name in CIF is   ',name

C....I.. List the audit record (possible text line sequence)
         write(6,'(a)') ' Audit record'
140      f1 = char_('_audit_update_record', line)
         write(6,'(a)') line
         if(text_)  goto 140

C....J.. Extract atom site data in a loop
         write(6,'(/a)') ' Atom sites'
160      f1 = char_('_atom_site_label', name)
         f2 = numb_('_atom_site_fract_x', x, sx)
         f2 = numb_('_atom_site_fract_y', y, sy)
         f2 = numb_('_atom_site_fract_z', z, sz)
         f3 = numb_('_atom_site_U_iso_or_equiv', u, su)
         write(6,'(1x,a4,4f8.4)') name,x,y,z,u
         if(loop_)  goto 160
C
         goto 120
200      continue
         end
```

Fig. 5.4.9.1. Sample program *CIF_IN*. See text for explanation.

`'#< not in dictionary'`. This applies to both looped and single data items.

A more complex example of writing a CIF is given in the program *cif2cif* available with the *CIFtbx* release. A similar program that reads a CIF and writes an XML file is *cif2xml*, also available with the *CIFtbx* release.

5.4.11. Error-message glossary

The *CIFtbx* routines will generate explicit error messages in the printout or in the created CIF if requested to do so (*e.g.* during dictionary checks). If data processing cannot continue (*i.e.* fatal errors), an appropriate error message is placed in the printout and execution terminates. However, the default approach is to remain mute and for error detection to be monitored by the application program *via* the *CIFtbx* functions returning `.true.` or `.false.` values that tell the application program whether the command was performed correctly. This places the primary responsibility for error checking on the application software. The importance of this approach is that it enables the local application to respond to run-time problems in a controlled way and to take corrective action if it is possible. However, some types of processing errors, such as exceeding the dimensions of critical *CIFtbx* arrays, do require appropriate messages to be issued and for execution to cease.

```
data_mumbo_jumbo

_audit_creation_date              91-03-20
_audit_creation_method            from_xtal_archive_file_using_CIFIO
_audit_update_record
; 91-04-09          text and data added by Tony Willis.
  91-04-15          rec'd by co-editor with diagram as manuscript HL7
;
_dummy_test     "rubbish to see what dict_ says"

_chemical_name_systematic
    trans-3-Benzoyl-2-(tert-butyl)-4-(iso-butyl)-1,3-oxazolidin-5-one
_chemical_formula_moiety           'C18 H25 N O3'
_chemical_formula_weight           303.40
_chemical_melting_point            ?

####_cell_length_a                    5.959(1)
_cell_length_b                     14.956(1)
_cell_length_c                     19.737(3)
_cell_measurement_theta_min        25
_cell_measurement_theta_max        31
_symmetry_cell_setting             orthorhombic

loop_
_atom_site_label
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
_atom_site_U_iso_or_equiv
_atom_site_thermal_displace_type
_atom_site_calc_flag
  s  .20200  .79800  .91667  .030(3)  Uij  ?
  o  .49800  .49800  .66667  .02520  Uiso  ?
  c1  .48800  .09600  .03800  .03170  Uiso  ?

loop_ _blat1 _blat2 1 2 3 4 5 6 a b c d 7 8 9 0
```

Fig. 5.4.9.2. Example CIF read by the sample program *CIF_IN* shown in Fig. 5.4.9.1.

```
 CIFtbx warning: test.cif data_mumbo_jumbo line:     8
 Data name _dummy_test not in dictionary!
 CIFtbx warning: test.cif data_mumbo_jumbo line:     35
 Data name _blat1 not in dictionary!
 CIFtbx warning: test.cif data_mumbo_jumbo line:     35
 Data name _blat2 not in dictionary!

 Access items in data block  mumbo_jumbo

 Cell dimension(s) missing!
 Cell     0.0000   14.9560   19.7370   0.0000    0.0010    0.0030
 Cell setting   orthorhombic

 Next data name in CIF is   _atom_type_symbol

 Audit record
  91-04-09          text and data added by Tony Willis.
  91-04-15          rec'd by co-editor with diagram as manuscript HL7

 Atom sites
 s     0.2020  0.7980  0.9167  0.0300
 o     0.4980  0.4980  0.6667  0.0252
 c1    0.4880  0.0960  0.0380  0.0317
```

Fig. 5.4.9.3. Printout from the example program *CIF_IN* run on the test file of Fig. 5.4.9.2.

CIFtbx error messages are in four parts: 'warning' or 'error' header line, the name of the file being processed, the current data block or save frame, and the line number. Another line contains the text of the message.

5.4.11.1. Fatal errors: array bounds

The following fatal messages are issued if the *CIFtbx* array bounds are exceeded. Operation terminates immediately. Array bounds can be adjusted by changing the PARAMETER values in ciftbx.sys. If the value of MAXBUF needs to be changed, the file ciftbx.cmv must also be updated.

```
Input line_ value > MAXBUF
Number of categories > NUMBLOCK
Number of data names > NUMBLOCK
Cifdic names > NUMDICT
Dictionary category names > NUMDICT
Items per loop packet > NUMITEM
Number of loop_s > NUMLOOP
```

```
C....... Open a new CIF
400      if(pfile_('test.new'))  goto 450
         write(6,'(//a/)') ' Output CIF by this name exists already!'
         goto 500
C....... Request dictionary validation check
450      if(dict_('cif_core.dic','valid'))  goto 460
         write(6,'(/a/)') ' Requested Core dictionary not present'
C....... Insert a data block code
460      f1 = pdata_('whoops_a_daisy')
C....... Enter various single data items to show how
         f1 = pchar_('_audit_creation_method','using CIFtbx')
         f1 = pchar_('_audit_creation_extra2',"Terry O'Connell")
         f1 = pchar_('_audit_creation_extra3','Terry O"Connell')
         f1 = ptext_('_audit_creation_record',' Text data may be ')
         f1 = ptext_('_audit_creation_record',' entered like this')
         f1 = ptext_('_audit_creation_record',' or in a loop.')
         f1 = pnumb_('_cell_measurement_temperature', 293., 0.)
         f1 = pnumb_('_cell_volume', 1759.0, 13.)
         f1 = pnumb_('_cell_length_b', 8.75353553524313,0.)
         f1 = pnumb_('_cell_length_c', 19.737, .003)
C....... Enter some looped data
         f1 = ploop_('_atom_type_symbol')
         f1 = ploop_('_atom_type_oxidation_number')
         f1 = ploop_('_atom_type_number_in_cell')
         do 470 i=1,3
         f1 = pchar_(' ',alpha(1:i))
         f1 = pnumb_(' ',float(i),float(i)*0.1)
470      f1 = pnumb_(' ',float(i)*8.64523,0.)
C....... Do it again but as contiguous data with text data
         f1 = ploop_('_atom_site_label')
         f1 = ploop_('_atom_site_occupancy')
         f1 = ploop_('_some_silly_text')
         do 480 i=1,2
         f1 = pchar_(' ',alpha(1:i))
         f1 = pnumb_(' ',float(i),float(i)*0.1)
480      f1 = ptext_(' ',' Hi Ho the diddly oh!')
500      call close_
```

Fig. 5.4.10.1. Sample program to create a CIF.

```
data_whoops_a_daisy
_audit_creation_method      'using CIFtbx'
_audit_creation_extra2      'Terry O'Connell'    #< not in dictionary
_audit_creation_extra3      'Terry O"Connell'    #< not in dictionary
_audit_creation_record
;Text data may be
 entered like this
 or in a loop.
;
_cell_measurement_temperature   293
_cell_volume                    1759(13)
_cell_length_b                  8.75354
_cell_length_c                  19.737(3)
loop_
    _atom_type_symbol
    _atom_type_oxidation_number
    _atom_type_number_in_cell
        a               1.00(10)        8.64523
        ab              2.0(2)          17.2905
        abc             3.0(3)          25.9357
loop_
    _atom_site_label
    _atom_site_occupancy
    _some_silly_text                    #< not in dictionary
        a               1.00(10)
;Hi Ho the diddly oh!
;
        ab              2.0(2)
;Hi Ho the diddly oh!
;
```

Fig. 5.4.10.2. Sample CIF created by the example program of Fig. 5.4.10.1.

However, the message

`More than MAXBOOK bookmarks requested`

is not 'fatal', in the sense that the function `bkmrk_` returns `.false.` to permit appropriate action before termination. This is effectively a fatal error for which recompilation with a larger value of `MAXBOOK` is necessary. However, this is usually the result of a logic error in the application, and the error has been made non-fatal to allow the programmer to insert debugging code, if desired. The application should clean up and exit promptly.

5.4.11.2. Fatal errors: data sequence, syntax and file construction

`Dict_ must precede ocif_`
Dictionary files must be loaded before an input CIF is opened because some checking occurs during the CIF loading process.

`Illegal tag/value construction`
Data name (*i.e.* a 'tag') and data values are not matched (outside a looped list). This usually means that a data name immediately follows another data name, or a data value was found without a preceding data name. The most likely cause of this error is the failure to provide '**.**' or '**?**' for missing or unknown data values or a failure to declare a **loop_** when one was intended.

`Item miscount in loop`
Within a looped list the total number of data values must be an exact multiple of the number of data names in the **loop_** header.

`Prior save-frame not terminated`
Save-frame terminator found out of context. Save frames must start with **save_*framecode*** and end with **save_**. These messages will be issued if this does not occur.

`Syntax construction error`
Within a data block or save frame the number of data values does not match the number of data names (ignoring loop structures). This message should occur only if there is an internal logic error in the library. Normally the program will terminate on `Item miscount in loop` first.

`Unexpected end of data`
When processing multi-line text the end of the CIF is encountered before the terminal semicolon.

5.4.11.3. Fatal errors: invalid arguments

The following messages are generated by calls with invalid arguments.

`Call to find_ with invalid arguments`
`Internal error in putnum`

5.4.11.4. Warnings: input errors

`Category <cat-code> first implicitly defined in cif`
The category code in the DDL2 data name is not matched by an explicit definition in the dictionary. This may be intentional but usually indicates a typographical error in the CIF or the dictionary.

`Data name <name> not in dictionary!`
The data item name `<name>` was used in the CIF but could not be found in the dictionary.

`Data block header missing`
No **data_** or **global_** was found when expected.

`Duplicate data item <name>`
Two or more identical data names `<name>` have been detected in a data block or save frame.

`Exponent overflow in numeric input`
`Exponent underflow in numeric input`
The numeric value being processed has an exponent that cannot be processed on this machine. If the string involved is not intended as a number, then surrounding it with quotes may resolve the problem.

`Heterogeneous categories in loop <new cat-code>`
`  vs <old cat-code>`
Looped lists should not contain data items belonging to different categories. This error occurs if the category of a new data item fails to match the category of a prior data item. A special category (none) is used to denote item names for which no category has been declared. Warnings are not issued on this level for a loop for which all data items have no declared category.

`Input line length exceeds line_`
Non-blank characters were found beyond the value given by the variable `line_`. The default value for `line_` is 80 (optionally increased to 2048 in *CIFtbx* 2.7 and later for CIF 1.1 compliance). The extra characters in column positions `line_` + 1 through `MAXBUF` will be processed but the input file may need to be reformatted for use with other CIF-handling programs.

`Missing loop_ items set as DUMMY`
A looped list of output items was truncated with an incomplete loop packet (*i.e.* the number of items did not match the number of **loop_** data names). The missing values were set to the *character string* 'DUMMY'.

`Numb type violated <name>`
The data item `<name>` has been processed with an explicit dictionary type numb, but with a non-numeric value. Note that the values '**?**' or '**.**' will *not* generate this message.

`Quoted string not closed`
Character values may be enclosed by bounding quotes. The strict definition of a 'quoted-string' value is that it must start with a `<wq>` digraph and end with a `<qw>` digraph, where `w` is a white-space character blank or tab and `q` is a single or double quote, and the same type of quote mark is used in the terminal digraph as was used in the initial digraph. This message is issued if these conditions are not met.

5.4.11.5. Warnings: output errors

`Converted pchar_ output to text for <string>`
An attempt was made to write a string with `pchar_` instead of `ptext_`, but the string contains a combination of characters for which `ptext_` must be used.

`ESD less than precision of machine`
`Overflow of esd`
`Underflow of esd`
A call to `pnumb_` or `numb_` was made with values of the number and standard uncertainty (e.s.d) which cannot be presented properly on this machine. A bounding value of 0 or 99999 is used for the e.s.d.

`Invalid value of esdlim_ reset to 19`
In processing numeric output, a value of `esdlim_` less than 9 or greater than 99999 was found. `esdlim_` is then set to 19.

`Missing loop_ name set as _DUMMY`
`Missing loop_ items set as DUMMY`
In processing a **`loop_`**, a dummy string has been inserted for a missing header or value.

`Output CIF line longer than line_`
In outputting a line, the data exceed the limit specified in `line_`. This occurs only if a single data name or a value exceeds this limit.

`Out-of-sequence call to end text block`
The termination of a text block has been invoked before a text block has been started. This can only occur with irregular use of the *CIFtbx* routines rather than the standard interface routines.

`Output prefix may force line overflow`
A prefix string placed in `prefx_` exceeds `line_` less the allowed length of tags.

`Prefix string truncated`
A prefix string specified to `prefx_` is longer than the maximum line length allowed. The prefix string is truncated and processing continues.

5.4.11.6. Warnings: dictionary checks

`Aliases and names in different loops; only using first alias`
If a DDL2 dictionary contains a loop of alias declarations, the corresponding data-name declarations are expected to be in the same loop. Only the first alias name is used.

`Attempt to redefine category for item`
`Attempt to redefine type for item`
If a DDL2 dictionary contains a category or type for a data item that conflicts with an earlier declaration, the first is used.

`Categories and names in different loops`
`Types and names in different loops`
If a DDL2 dictionary contains a loop of category or type declarations, the corresponding data-name declarations are expected to be in the same loop. Only the first category name or type is used.

`Category id does not match block name`
In a DDL2 dictionary, the save-frame code is expected to start with the category name. If a category name within the frame is not within a loop, it is checked against that in the frame code and a warning is issued if these do not match.

`Conflicting definition of alias`
A DDL2 dictionary contains a new declaration of a data-name alias which is in conflict with a previous alias definition. The first alias declaration is used.

`Duplicate definition of same alias`
A DDL2 dictionary contains a new declaration of an alias for a data name which duplicates a previously defined alias pair.

`Item name <name> does not match category name`
If category checking is enabled and the category assigned to an item name does not match the intial characters of the item name, this message is issued. This may indicate a typographical error or a deprecated item in the dictionary.

`Item type <type-code> not recognised`
The DDL2 dictionary type codes are translated to the DDL type codes 'numb', 'char' and 'text'. If an unrecognized type code is found no translation occurs.

`Multiple DDL category definitions`
`Multiple DDL name definitions`
`Multiple DDL type definitions`
`Multiple DDL related item definitions`
`Multiple DDL related item functions`
DDL1 and DDL2 declarations for categories, data names, data types and related items are used in the same data block or save frame.

`Multiple categories for one name`
`Multiple types for one name`
A dictionary contains a loop of category or type definitions and an unlooped declaration of a single data name. The first category or type definition is used.

`No category defined in block <name> and name <name> does not match`
A DDL2 dictionary contains no category for the defined data item and it was not possible to derive an implicit category from the block name. This usually indicates a typographical error in the dictionary.

`No category specified for name <name>`
A dictionary contains categories and category checking is enabled but no category is defined for the named data item.

`No name defined in block`
`No name in the block matches the block name`
These messages are issued if a dictionary save frame or data block contains no name definition or if all the names defined fail to match the block name.

`No type specified for name <name>`
A type code is missing from a dictionary and type checking was requested in the `dict_` invocation.

`One alias, looped names, linking to first`
A DDL2 dictionary may contain a list of data names and a single alias outside this loop. In this case, the correct name to which to link the alias must be derived implicitly. If the save-frame code matches the first name in the loop no warning is issued, because the use of the block name was probably the intended result, but if no such match is found this warning is issued.

5.4.12. Internals and programming style

CIFtbx is programmed in a highly portable Fortran programming style. However, on some older systems, some adaptation may be necessary to allow compilation. Implementors should be aware of the extensive use of variables in common blocks to transmit information and control execution (programming by side-effects), the use of the `INCLUDE` statement, the use of the `ENDDO` statement, the names of routines used internally by the package, the use of names longer than six characters and the use of names including the underscore character.

Some aspects of the internal organization of the library to deal with characteristics of CIFs are worth noting. *CIFtbx* copies an input CIF to a direct-access (*i.e.* random-access) file, but writes an output CIF directly. All data names are converted to lower case to deal with the case-insensitive nature of CIF. A hierarchy of parsing routines is used to deal with processing white space.

The major issues of programming style and internals are summarized here. See the *Primer* on the CD-ROM for more information.

5.4.12.1. Programming style

A traditional Fortran style of programming is used in *CIFtbx*. Common blocks are declared to report and control the state of the processing. This allows argument lists to be kept short and avoids the need to create complex data structure types, but introduces extensive 'programming by side-effects'. In order to reduce the impact of this approach on users, two different views of the common blocks are provided. The declarations in ciftbx.cmn are needed by all users. The more extensive declarations in ciftbx.sys, which include the same common declarations as are found in ciftbx.cmn and additional declarations used internally within *CIFtbx*, are provided for use in maintaining the library. Caution is needed in making internal modifications to the library to maintain the desired relationships among the actions of various routines and the states of variables declared in the common blocks.

Statements are written in the first 72 columns of a line, reserving columns one through five for statement labels and using column six for continuation. Approaches that would require the use of C libraries or non-portable Fortran extensions are avoided. For this reason, all the internal service routines are written in Fortran, all memory needed is preallocated with DIMENSION statements and a direct-access file is used to hold the working copy of a CIF.

5.4.12.2. Memory management

Since *CIFtbx* does static memory allocation with DIMENSION statements, it is sometimes necessary to adjust the array dimensions chosen to suit a particular application. It may also be necessary to increase the storage allocated for individual tags to allow for unusually long ones.

The sizes of most arrays and strings used in *CIFtbx* that might require adjustment are controlled by PARAMETER statements in the files ciftbx.sys and ciftbx.cmv (the variable-declaration portion of ciftbx.cmn). The parameters are shown in Table 5.4.12.1.

These values can result in *CIFtbx* requiring more than a megabyte of memory. On smaller machines working with a small dictionary and simple CIFs, considerable space can be saved by reducing the values of NUMDICT and NUMBLOCK.

On the other hand, an application working with several layered dictionaries and large and complex CIFs with many data items and many loops in a data block might require a version of *CIFtbx* with larger values of NUMDICT, NUMBLOCK and, perhaps, of NUMLOOP.

The variables NUMPAGE and NUMCPP control the amount of memory to be used to buffer the direct-access file and the size of the data transfers to and from that file. Smaller values will reduce the demand for memory at the expense of slower execution.

5.4.12.3. Use of INCLUDE

The INCLUDE statement allows the statements in the specified file to be treated as if they were being included in a program in place of the INCLUDE statement itself. This simplifies the maintenance of common-block declarations and is an important tool in keeping code well organized. In *CIFtbx*, the INCLUDE statement is used to bring the statements in the files ciftbx.cmn and ciftbx.sys into programs where they are needed, and to simplify ciftbx.cmn and ciftbx.sys by using INCLUDEs of the files ciftbx.cmv and ciftbx.cmf. The file ciftbx.cmv contains the definitions of the essential *CIFtbx* data structures as common blocks, for inclusion in both ciftbx.cmn for user applications and in ciftbx.sys for the *CIFtbx* library routines themselves. Most compilers handle the INCLUDE statement, but, if necessary, a user may replace any or all of the INCLUDE statements with the contents of the indicated file. For example, the only non-comments in ciftbx.cmn are

Table 5.4.12.1. *Parameter statements in CIFtbx*

NUMCHAR	Maximum number of characters in data names (default 48)
MAXBUF	Maximum number of characters in a line (default 200, increased to 4096 in *CIFtbx* 2.7 and later)
NUMPAGE	Number of memory resident pages (default 10)
NUMCPP	Number of characters per page (default 16 384)
NUMDICT	Number of entries in dictionary tables (default 3200)
NUMHASH	Number of hash table entries (a modest prime, default 53)
NUMBLOCK	Number of entries in data-block tables (default 500)
NUMLOOP	Number of loops in a data block (default 50)
NUMITEM	Number of items in a loop (default 50)
MAXTAB	Maximum number of tabs in output CIF line (default 10)
MAXBOOK	Maximum number of simultaneous bookmarks (default 1000)

```
include  'ciftbx.cmv'
include  'ciftbx.cmf'
```

This means that the file ciftbx.cmn could be replaced by a concatenation of the two files ciftbx.cmv and ciftbx.cmf.

5.4.12.4. Use of ENDDO

CIFtbx makes some use of the ENDDO statement (as well as nested IF, THEN, ELSE, ENDIF constructs) to improve readability of the source code. Most compilers accept the ENDDO statement, but if conversion is needed then constructs of the form

```
      do index = istart, iend, incr
        ...
      enddo
```

should be changed to

```
      do nnn index = istart, iend, incr
        ...
nnn   continue
```

where *nnn* is a unique statement number, not used elsewhere in the same routine.

5.4.12.5. Names of internal routines

The following routines are used internally by later versions of *CIFtbx*. If these names are needed for other routines, then changes in the library will be needed to avoid conflicts.

(*a*) Variable initialization:

```
block data
```

Critical *CIFtbx* variables are initialized with data statements in a block data routine.

(*b*) Control of floating-point exceptions:

```
subroutine clearfp
```

If a system requires special handling of floating-point exceptions, the necessary calls should be added to this subroutine.

(*c*) Message processing:

```
subroutine err (mess)
  character mess*(*)
subroutine warn (mess)
  character mess*(*)
subroutine cifmsg (flag, mess)
  character mess*(*), flag*(*)
```

Error and warning messages are processed through these three routines.

(*d*) Internal service routines:

```
subroutine dcheck
  (name, type, flag, tflag)
  logical flag, tflag
  character name*(*), type*4
subroutine eotext
subroutine eoloop
subroutine excat
  (sfname, bcname, lbcname)
  character*(*) sfname, bcname
  integer lbcname
subroutine getitm (name)
  character name*(*)
subroutine getstr
subroutine getlin (flag)
  character flag*4
subroutine putstr (string)
  character string*(*)
```

These routines are used internally by the library. The subroutine `dcheck` validates names against dictionaries. The subroutines `eotext` and `eoloop` are used to ensure termination of loops and text strings. The subroutines `getitm`, `getstr` and `getlin` extract items, strings and lines from the input CIF. The subroutine `putstr` writes strings to the output CIF.

(*e*) Numeric routines:

```
subroutine ctonum
subroutine putnum (numb, sdev, prec)
  double precision numb, sdev, prec
```

The routine `ctonum` converts a string to a number and its standard uncertainty. The subroutine `putnum` converts a number and standard uncertainty to an output string.

(*f*) String manipulation:

```
subroutine detab
integer function lastnb (str)
  character str*(*)
character*(MAXBUF) function locase (name)
  character name*(*)
```

The subroutine `detab` converts tabs to blanks. The function `lastnb` finds the column position of the last non-blank character in a string. The function `locase` converts a string to lower case.

(*g*) Hash-table processing:

```
subroutine hash_find (name, name_list,
  chain_list, list_length, num_list,
  hash_table, hash_length, ifind)
  character name*(*),
    name_list(list_length)
  integer hash_length,
    chain_list(list_length),
    hash_table(hash_length), ifind
subroutine hash_store (name, name_list,
  chain_list, list_length, num_list,
  hash_table, hash_length, ifind)
  character name*(*),
    name_list(list_length)
  integer hash_length,
    chain_list(list_length),
    hash_table(hash_length), ifind
integer function hash_value (name, hash_length)
  character name*(*)
  integer hash_length
```

These routines are used to manipulate the internal hash tables used by the library.

5.4.12.6. Use of the underscore character

All the externally accessible *CIFtbx* commands and variables terminate with the underscore character. This works well on most systems, but can cause occasional problems, because traditional Fortran does not include the underscore in the character set and some operating systems reserve the underscore as a system flag, for example to distinguish C-language library routines from those written in Fortran. If conversion is needed, and the local compiler allows long variable and subroutine names, then the simplest approach would be to make a local variant of *CIFtbx* in which every occurrence of underscore in a function, subroutine or variable name is changed to a distinctive character pattern (*e.g.* 'CIF' or 'qq'), but caution is needed, since there are many *character string*s used in the library that include the underscore. For example, in changing the variable `loop_` to loopCIF, it would be a mistake to change the statement

```
if(strg_(1:5).eq.'loop_')
  type_='loop'
```

to

```
if(strg_(1:5).eq.'loopCIF')
  type_='loop'
```

5.4.12.7. Names longer than six characters

CIFtbx uses some function, subroutine and variable names longer than six characters to improve readability, but, in most cases, consistent truncation of all uses of a name to six characters will not cause any problems.

5.4.12.8. File management

CIFtbx allows the user to read from one CIF while writing to another. The input CIF is first copied to a direct-access file to allow random access to desired portions of the input CIF. Since CIF allows data items to be presented in any order, the alternatives to the use of a direct-access file would have been to create memory-resident data structures for the entire CIF or to track position and make multiple search passes through the file as data items are requested. When programming for personal and laboratory computers with limited memory and which may lack virtual memory capabilities, assuming the availability of enough memory for large CIFs would greatly restrict the applications within which *CIFtbx* could be used. However, the disk accesses involved in using a direct-access file slow execution. When working on larger computers, execution speed can be increased at the expense of memory by increasing the number of memory-resident pages (see the parameter `NUMPAGE` above). If the number of pages times the number of characters per page (`NUMCPP`) is large enough to hold the entire CIF, the application will run much faster.

Direct reading of the input CIF, making multiple passes when data items are requested in a different order to that in which they are presented in the CIF, is only practical when the number of out-of-order requests is small and the applications will not need to be used as a filter, perhaps reading the output of another program 'on-the-fly'. Since we cannot predict the range of applications and CIFs for which *CIFtbx* will be used, and direct reading could become impossibly slow, *CIFtbx* uses a direct-access file.

The processing of an output CIF is simpler than reading a CIF. The application determines the order in which the writing is to be done. No sorting is normally needed. Therefore *CIFtbx* writes an output CIF directly.

5.4.12.9. Case sensitivity

A CIF may contain data names in upper, lower or a mixture of cases. Internally, *CIFtbx* does all its name comparisons in lower case, using the function `locase` (see above) to convert. Good style, however, dictates the use of certain case combinations in certain names. Therefore *CIFtbx* does this lower-case conversion as needed, preserving the original case for whatever use may be desired. An application needing maximum speed and which does not need to preserve the cases in the original CIF might consider doing the case conversion once and removing the use of `locase`.

5.4.12.10. Management of white space

CIF does not care about white space. One blank or tab is equivalent to many blanks or tabs or empty lines in separating data names from values and values from one another. The internal routine `getstr` extracts the next white-space-delimited string, using `getlin` to deliver input lines from the direct-access file as required. Since Fortran does not provide dynamic memory allocation, this approach presents a problem with multi-line text fields. Rather than allocate a large fixed space that might not hold still larger text fields, the library delivers those strings one line at a time. As with case sensitivity, *CIFtbx* does white-space scanning repeatedly, keeping the original presentation (including tabs) available should an application need access to it. The author of an application needing maximum speed, not needing the original presentation and wishing to conserve disk space might wish to modify the operation of *CIFtbx* to remove all comments and compress all separating white space to single blanks or line terminators in an initial sweep.

5.4.13. Distribution

Version 2.6.4 and an early release of version 3 of *CIFtbx* are included on the accompanying CD-ROM. As later versions are developed they will be available from the IUCr (http://www.iucr.org/iucr-top/cif) and authors' (http://www.bernstein-plus-sons.com/software/ciftbx) web sites.

The release kit is a compressed C-shell archive ciftbx.cshar.Z or a compressed shell archive ciftbx.shar.Z. Only one is needed. The uncompressed files ciftbx.cshar or ciftbx.shar are needed for implementation.

We are grateful to Frances C. Bernstein for her helpful comments and suggestions.

References

Bernstein, F. C. & Bernstein, H. J. (1996). *Translating mmCIF data into PDB entries. Acta Cryst.* A**52** (Suppl.), C576. Software available at http://www.bernstein-plus-sons.com/software/cif2pdb.

Bernstein, H. J. (1997). *cif2cif. CIF copy program*. Bernstein + Sons, Bellport, NY, USA. Included in http://www.bernstein-plus-sons.com/software/ciftbx.

Bernstein, H. J., Bernstein, F. C. & Bourne, P. E. (1998). *CIF applications. VIII. pdb2cif: translating PDB entries into mmCIF format. J. Appl. Cryst.* **31**, 282–295. Software available at http://www.bernstein-plus-sons.com/software/pdb2cif.

Bernstein, H. J. & Hall, S. R. (1998). *CIF applications. VII. CYCLOPS2: extending the validation of CIF data names. J. Appl. Cryst.* **31**, 278–281. Software available at http://www.bernstein-plus-sons.com/software/ciftbx/cyclops.src.

Bray, T., Paoli, J. & Sperberg-McQueen, C. M. (1998). *Extensible Markup Language (XML)*. W3C recommendation 10-February-1998. http://www.w3.org/TR/1998/REC-xml-19980210.

Hall, S. R. (1993*a*). *CIF applications. II. CIFIO: for CIF input/output in the Xtal system. J. Appl. Cryst.* **26**, 474–479.

Hall, S. R. (1993*b*). *CIF applications. IV. CIFtbx: a tool box for manipulating CIFs. J. Appl. Cryst.* **26**, 482–494.

Hall, S. R. & Bernstein, H. J. (1996). *CIF applications. V. CIFtbx2: extended tool box for manipulating CIFs. J. Appl. Cryst.* **29**, 598–603.

Keller, P. A. (1996). *A mmCIF toolbox for CCP4 applications*. *Acta Cryst.* A**52** (Suppl.), C576.

Murray-Rust, P. & Rzepa, H. (1999). *Chemical markup, XML and the Worldwide Web. 1. Basic principles. J. Chem. Inf. Comput. Sci.* **39**, 928–942.