PRIMER PLUS

by

Herbert J. Bernstein

and

Sydney R. Hall

Preface

The Crystallographic Information File (CIF) format is one of the most commonly used electronic data handling protocols in chemistry and crystallography for exchanging and archiving structural and diffraction information. Because the CIF syntax requirements and data definitions are coordinated and supported by the International Union of Crystallography as part of their publishing and archival activities, there is established support for this format both in database entry and in retrieval tasks. This usage has spawned the need to develop more comprehensive software for generating, reading and manipulating CIF data.

This book is an instruction and reference manual for programmers employing the *CIFtbx* library of Fortran functions to develop CIF applications. Subject to the conditions specified in Appendix A, the *CIFtbx* library is freely available software for non-commercial use. Commercial software developers must seek written permission from the authors before applying this software. *CIFtbx,* and associated CIF data definition dictionaries, may be obtained from various web sites (see Appendix B for installation instructions).

The *CIFtbx* library and this manual are intended for both novices and experts of CIF applications. The toolbox has already been used in the development of CIF manipulation programs such as *Cyclops* [Bernstein, Hall 97], *CIFIO* [Hall 93a], *cif2cif* [Bernstein 97a], *pdb2cif* [Bernstein, Bernstein, Bourne 98] and *cif2pdb* [Bernstein, Bernstein 96]. Extracts from some of these applications are used herein to illustrate various programming approaches.

This edition of the manual is for use with *CIFtbx* version 2.6. Scientific papers on *CIFtbx* [Hall 93c, Hall, Bernstein 96] provide background information on earlier versions of the tool box but lack the detail of a primer and reference manual. The *CIFtbx* tools described in this manual are appropriate for all current CIF applications and dictionaries. This includes the access and application of data definitions in dictionaries based on the definition language DDL1 [Hall, Cook 95], and on the extended language DDL2 [Westbrook, Hall 95] such as the macromolecular dictionary [Fitzgerald, Berman, Bourne, McMahon, Watenpaugh, Westbrook, 96].

The first two chapters of the manual introduce the general concepts of the CIF syntax and are intended for programmers who have no prior knowledge of this format. This is the initial *primer* information. Later chapters give detailed explanations on the 21 functions and 36 variables that make up the tools, and how they are applied to simple and complex tasks. The appendices at the end of the manual explain how to implement the tool box software on your computer, and provide background information on the DDL used to define CIF data items, and to construct CIF dictionaries.

CONTENTS

Recent History and Acknowledgements

The CIF format was first adopted by the IUCr for journal submissions in 1990, following the publication of the CIF core dictionary [Hall, Allen & Brown, 90]. Since then there has been continual growth in the use of CIFs and in the development of software for CIF generation and manipulation. In the past two years there have been appreciable changes to the nature of CIF applications. These have been brought about largely because of new data definitions in the macromolecular CIF dictionary [Fitzgerald, Berman, Bourne, McMahon, Watenpaugh, Westbrook 96 and Bourne, Berman, McMahon,. Watenpaugh, Westbrook, Fitzgerald 96]. This, and the powder diffraction dictionary [Toby 97], were recently adopted by the IUCr (June 1997) as standards for exchanging crystallographic data in these fields. The adoption of the macromolecular dictionary, in particular, signals a watershed in the way that this type of structural data will be handled in the future.

The most recent impetus for newer and more versatile versions of *CIFtbx* has been to assist one of us (HJB) in using CIF data derived from Protein Data Bank files [Bernstein, Koetzle, Williams, Meyer, Brice, Rodgers, Kennard, Shimanouchi, Tasumi 77]. In the development of *pdb2cif* [Bernstein, Bernstein, Bourne 98], *CIFtbx2* enabled hundreds of CIF data names, embedded in existing software, to be mapped into the DDL2 format, and for the existence of these items to be checked. *CIFtbx2* has been used in the recent release of the *Xtal 3.5 System* [Hall, King, Stewart 95], and in the upgrade of *CYCLOPS* to *CYCLOPS2* [Hall, Bernstein 96] (see Appendix C). It has provided the platform for the creation of *cif2cif*, a program which checks and reformats CIFs. *CIFtbx2* was used for rapid adaptation of a command-line driven lattice identification program to CIF [Bernstein, Andrews 96]

A primary objective with this toolbox has been to preserve the functionality of all dictionaries written the core dictionary language DDL1 while providing a seamless link to the richer DDL2 dictionaries. During this development we have leaned heavily on the cooperation of our colleagues and collaborators. Many people have contributed to the CIF development and although we are certain to not mention many workers who have given valuable help at some stage, we must highlight the special recent efforts of Helen Berman, Frances Bernstein, Phil Bourne, Paula Fitzgerald, Brian McMahon and John Westbrook.

CHAPTER 1

What is a CIF?

1.1. Introduction

What is a CIF? To a crystallographer or a structural chemist, it is a simple and flexible way of storing and exchanging numerical or text data electronically. The letters C-I-F stand for *Crystallographic Information File* [Hall, Allen, Brown 91]. A CIF is a text file that can be easily read by humans or computers because of its very simple format. The rules governing this format are a subset of the general syntax of the Self-Defining Text Archive and Retrieval (STAR) File [Hall, 91].

The CIF format is extremely flexible. Data items may be placed anywhere in a file or a line, and in any order, provided that each data value is preceded by an identifying label. Here is an extract from a CIF. The data values are in bold type and the data identifiers (or names) are strings starting with an underscore.

```
_crystal_habit          irregular_tetrahedron
_crystal_colour         'blue green'
_crystal_density        1.765(4)

loop_
    _crystal_face_index_h
    _crystal_face_index_k
    _crystal_face_index_l
    _crystal_face_dist_from_centre   # in millimetres
          1    1    1   0.25
         -1   -1    1   0.27
          1   -1   -1   0.25
         -1    1   -1   0.29

_crystal_preparation
;  The compound is crystallised from ethanol by slow
evaporation.
;
```

1.2. Basic syntax

The above example illustrates many of the basic principles of a CIF.

1. All contents are *ascii* text.

2. Each *data value* (shown above in bold) must be preceded by an identifying *data name.*

3. A *data name* (or tag) is a character string starting with an underscore character.

4. *Data values* are of three basic types: number strings, character strings and text strings.
   - A *number string* may be in integer, decimal or scientific notation. Numbers may have an error estimate appended within parentheses (see `_crystal_density` above), if this is allowed by the data definition (see DDL description below).
   - A *character string* is a sequence of characters that is not a number; not preceded by an underscore, and does not exceed one 80 characters in length. If the string contains blanks it must be surrounded by quote characters (see `_crystal_habit`), in which case the string must not exceed 78 characters in length.
   - A *text string* may be one or more lines in length and must be bounded by semicolons in column 1 preceding the first character, and following the last character (see `_crystal_preparation`). Unless the data definition imposes more restrictive rules, a text string may be used any place where a character string might be expected.

5. Lists of repeated data values are to be preceded by data names in matching order. Such lists must be preceded by a `loop_` command.

6. A data name and its value (i.e. *tag/value pair*, or *tuple*) are referred to as a *data item*. Data items are grouped into *data blocks*. A data block is preceded by a `data_`<*name*> command. The <*name*> string is referred to as the *data block name* and this must be unique within a CIF.

7. Within a data block, each data name must be unique.

8. A CIF is restricted to 80-character lines.

9. The hash character '#' is used to start a comment on a line.

1.3. Case sensitivity.

Data names are **not** sensitive to the case of letters. For example, the strings

```
_ATOM_SITE_CARTN_X
_atom_site_cartn_x
_AtOm_SiTe_CaRtN_x
```

all represent the identical data name in a CIF. Strings that are not data names are case sensitive in that the case of letters must always be preserved.

1.4. Special characters

Certain characters in a CIF serve a special function when used in a particular way. A brief summary of these is given below. For more detail see [Hall, Spadaccini 95].

_ the underscore (underline) is used to start a data name, or to end of a command string, such as `loop_`. They terminate *CIFtbx* function and variable names. They sometimes are used to replace blanks in strings so as to avoid surrounding quotes.

<*w*> "white-space" characters such as blanks, tabs and end-of-lines are used to delimit fields in a CIF, i.e. one or more white-space characters serve to separate data names and values, provided the data names and values are not inside a quoted string (as with the `_crystal_colour` value above) or a text string (as with the `_crystal_preparation` value above).

the hash mark (sharp) disables syntactic processing of characters following on a line, except within a quoted or text string. The hash is used for comments in a CIF.

' the single quote (apostrophe) may be used to protect a character string, but not a number or text string, from internal syntactic processing. This is done by surrounding a character sequence with quote characters. More precisely the string must start with the digraph <*w*>' and end with the digraph '<*w*>. Within such a string characters such as _, <*w*>, # and " do not have special properties. Note that the ' character may also be placed within this string provided that it is not immediately trailed by a <*w*> character. The character string must not span multiple lines.

" the double quote serves the same function as **'**.

; the semicolon, if used as the first character in a line, is used to start and finish a sequence of lines, referred to as a string of type *text*.. The sequence newline-semicolon serves very much the same purpose as the single and double quotes, but is the only way to provide multiple line text as a value.

. the period character has a special meaning when used by itself as a data value. It usually means "the default value".

? the question mark character has a special meaning when used by itself as a data value. It usually means "value unknown".

$ the dollar sign is normally NOT a special character in a CIF. However, if a CIF contains *save frames* [Hall & Spadaccini, 95], the dollar sign is used at the start of save frame names when referred to as data values. To avoid confusion, do not use an unquoted dollar sign as a value in a CIF.

1.5. Syntax control words

Special words in a CIF control and signal syntax changes. These words can be easily recognised by a trailing underscore character.

`data_` signals the start of a new data block. A name is appended to this string (e.g. `data_crystal_description`). Each data block name within a CIF must be unique.

`loop_` signals the start of a repeated list of data items. `loop_` is followed by the data names of all items in the list. Then come the data values, in the same order as the data names, and these are repeated until another data name or control string is encountered.

`global_` signals the start of a new *global* data block. This serves the same function as `data_` , except that it contains items are assumed to be "global" rather than specific to a particular data block. Global data blocks do not have block names.

`stop_` signals the end of a *nested* list. Nested lists are not currently used in CIF's but they are in a STAR File [Hall & Spadaccini, 95]. An example of this is shown later.

`save_` signals the start, and the end, of a *save frame*. Save frames are not used in CIF's, but they are in DDL2 dictionaries. A save frame is used as a "macro" within a data block to contain, one or more data items. A save frame is "addressable" via a *frame code,* and each data name within a frame must be unique. A data block may contain any number of save frames. Because the identity of data items within a save frame is "protected" from items outside this frame, the same data name may be used in the data block or in other save frames. The `save_` string at the start a frame has an appended code that is unique within the data block. This code, preceded by a dollar character, i.e. $<code>, may be referred to as a data value, so as to "point to" a specific frame of data items. The `save_` string closing a frame does not have a code attached.

For a more formal description of STAR syntax, see Appendix D.

1.6. File examples

Some data file examples will be now used to illustrate syntax requirements.

1.6.1 A typical structural CIF

Here is an abbreviated version of typical CIF.

```
data_xtest2
|
_chemical_name_systematic
     hexamethyl-4,8-dioxaundecanedioate)bis(pyridine)dirhodium
_chemical_formula_sum            'C40 H62 N2 O12 Rh2 '
_chemical_formula_moiety         ?
_chemical_formula_weight         498.35
_symmetry_cell_setting           triclinic
_symmetry_space_group_name_H-M           'P -1'
|
loop_
_symmetry_equiv_pos_as_xyz
'x,y,z'              '-x,-y,-z'
|
_cell_length_a                   8.586(8)
_cell_length_b                   15.286(11)
_cell_length_c                   15.606(8)
_cell_angle_alpha                94.57(4)
_cell_angle_beta                 92.31(4)
_cell_angle_gamma                100.58(4)
_cell_formula_units_Z            4
_cell_volume                     2004(3)
```

Note the following in this example.

- The alignment of character strings in this (and any other) CIF is largely a matter of taste. Changing the white space between data names or values does not affect the meaning of the data; nor does any reordering of the items. The term "item" refers to a tag / value pair.

- The quotes are not needed for the `_symmetry_equiv_pos_as_xyz` values because they contain no embedded blanks, however, their presense does not alter the value. Because of embedded blanks in the formula, the quotes bounding the `_chemical_formula_sum` string are required. Double quotes would have worked as well.

- Because the value of `_chemical_formula_moiety` is unknown, its value is shown as a question mark. This item (i.e. the tag and the value) could have been omitted from the file, however, it is often convenient to retain the data name of a missing value as a reminder that it needs to be added. One of the most common errors in a CIF is the omission of a missing value (i.e. using a blank field) as this violates the requirement to match tags to values.

1.6.2 A STAR File

Here is an example of a STAR File to illustrate its much more extensive syntax. This file contains quantum chemical data on the water molecule. One can see that a STAR file in most respects is identical to a CIF file.

```
data_water

_qchem_chemical_name_common            water
_qchem_chemical_name_IUPAC             'oxygen dihydride'
_qchem_chemical_formula                'H2 O'

loop_
    _qchem_molecular_site_number
    _qchem_molecular_site_label
    _qchem_molecular_site_symbol
    _qchem_molecular_site_x
    _qchem_molecular_site_y
    _qchem_molecular_site_z
    _qchem_molecular_site_mass

    1  O1  O   0.00000  0.00000  0.00000  15.994915
    2  H1  H   0.00000  0.75753  0.58707   1.007825
    3  H2  H   0.00000 -0.75753  0.58707   1.007825

_qchem_molecular_mass_centre_x         0.0000000
_qchem_molecular_mass_centre_y         0.0000000
_qchem_molecular_mass_centre_z         0.0657023

loop_
    _qchem_basis_set_atom_name
    _qchem_basis_set_atom_symbol
    _qchem_basis_set_contraction_scheme
    _qchem_basis_set_funct_per_contraction
    loop_
        _qchem_basis_set_function_code
        _qchem_basis_set_function_count
        _qchem_basis_set_function_exponent
        _qchem_basis_set_function_coefficient

oxygen    O       (9,5,1)->[4,2,1]   {6:1:1:1,4:1,1}

       s 1     7816.540000      0.002031
       s 1     1175.820000      0.015436
       s 1      273.188000      0.073771
#.....................................data omitted for space
       d 7        0.900000      1.000000 stop_

hydrogen   H      (4,1)->[2,1]      {3:1,1}

       s 1        19.240600     0.032828
       s 1         2.899200     0.231208
       s 1         0.653400     0.817238
       s 2         0.177600     1.000000
       p 3         1.000000     1.000000 stop_
```

```
loop_
    _qchem_bond_site_label_1
    _qchem_bond_site_label_2
    _qchem_bond_distance_au
    _qchem_bond_distance
                            O1  H1   1.811095991  0.958390452
                            O1  H2   1.811095991  0.958390452

loop_
    _qchem_angle_site_label_1
    _qchem_angle_site_label_2
    _qchem_angle_site_label_3
    _qchem_angle
                          H1  O1  H2  104.44991917

_qchem_molecule_number_atoms                3
_qchem_molecule_number_electrons           10
_qchem_molecule_number_contractions        13
_qchem_molecule_charge                      0
_qchem_molecule_state_multiplicity          1
_qchem_molecule_occup_orb_doub              5
_qchem_molecule_occup_orb_sing_alpha        0
_qchem_molecule_occup_orb_sing_beta         0

_qchem_option_converge_criterion            1.0E-05
_qchem_option_variable_level_shift          yes


_qchem_calc_energy_electronic               -85.230179266
_qchem_calc_energy_nuclear                    9.183706230
_qchem_calc_energy_total                    -76.046473036
```

Note the following difference between this STAR file and a CIF.

- The _qchem_basis_set_ items in this STAR file are in nested loop. The _qchem_basis_set_function_ items are in a level 2 loop. Note that following the last set (or packet) of data values for these items there is a stop_ signal. This causes the nesting to revert to level 1.

1.7 Data definitions

CIF data items used in global data exchange applications, such as in archiving or publication, are usually defined in an electronic dictionary that has been formally approved by the IUCr. In that way, those that generate CIF data have a common understanding of what the data means with those that subsequently read that data. The definition of data items has become quite rigorous as a consequence of this requirement and involves special definition protocols that are incorporated in a dictionary definition language (DDL).

Each data definition needs to specify the function of an item, and list its particular characteristics or attributes. For instance, the definition needs to specify if an item is a number or a character string. Although very few users of CIF data need to understand how dictionaries and the individual definitions are constructed, a programmer writing CIF applications will benefit greatly by

knowing about the two types of DDL currently in use, appreciating the types of information contained within the DDL definitions, and understanding how it can be employed to validate data.

The format of all CIF electronic dictionaries conform to the STAR syntax, and may also be parsed with *CIFtbx* tools. In fact, the toolbox provides a specific function to read and cross check attributes from dictionaries.

Existing dictionaries are written using two different DDL's. DDL1 has been used to construct the CIF Core, Powder and several other dictionaries. A more recent dictionary language, DDL2, is used to specify the macromolecular dictionary mmCIF [Fitzgerald, Berman, Bourne, McMahon, Watenpaugh, Westbrook 96].

1.7.1 DDL1 definition examples

1.7.1.1 DDL1 example 1

Here is DDL1 definition of the data items `_atom_site_fract_x`, `_atom_site_fract_y`, and `_atom_site_fract_z` from the Core dictionary.

```
data_atom_site_fract_
    loop_ _name                '_atom_site_fract_x'
                               '_atom_site_fract_y'
                               '_atom_site_fract_z'
    _category                    atom_site
    _type                        numb
    _type_conditions             esd
    _list                        yes
    _list_reference            '_atom_site_label'
    _enumeration_default         0.0
    _definition
;             Atom site coordinates as fractions of the _cell_length_
              values.
;
```

The precise meanings of the different DDL1 attributes such as `_name`, `_category`, etc. are given in [Hall, Cook 95] and [McMahon 95].

Note the following in this definition.

- when data items form an irreducible set, such as with the fractional coordinates x, y, and z, or the diffraction indices h, k, and l, they are defined in the same DDL1 data block. In DDL2 each data item is defined separately.

- the `_list` attribute tells us that the fractional coordinates must be present in a looped list of category `atom_site`.

- the `_list_reference` attribute specifies that the data item `_atom_site_label` must be present in the same looped list as the fractional coordinates for the list of category `atom_site` items to be valid.

- the _type_conditions attribute places the condition *esd* on the _type value of *numb*. This means that fractional coordinate numbers may have the estimated standard deviation (i.e. standard uncertainty) values appended within parentheses.

- the _enumeration_default attribute defines the value that a fractional coordinate is assumed to have, if it is missing from a CIF.

1.7.1.2 DDL1 example 2

Here is the DDL1 definition of the data item _atom_site_label. This is the item referred to above as the _list_reference data that must be present in a list of items of *category* atom_site, in order that the CIF be valid.

```
data_atom_site_label
    _name                       '_atom_site_label'
    _category                     atom_site
    _type                         char
    _list                         yes
    _list_mandatory               yes
    loop_ _list_link_child      '_atom_site_aniso_label'
                                '_geom_angle_atom_site_label_1'
                                '_geom_angle_atom_site_label_2'
                                '_geom_angle_atom_site_label_3'
                                '_geom_bond_atom_site_label_1'
                                '_geom_bond_atom_site_label_2'
    loop_ _example                C12     Ca3g28     Fe3+17     H*251
                                  boron2a   C_a_phe_83_a_0  Zn_Zn_301_A_0
    _definition
;             The _atom_site_label is a unique identifier for a
              particular site in the crystal.
;
```

Note the following in this definition.

- the attribute _list_mandatory with a value of yes signals that this item *must* be present in any list of category atom_site.

- the _list_link_child attributes specify data items that are 'child' dependencies of _atom_site_label. This means that this item *must* be present in the CIF if any of the dependent items is present.

- the _list_reference attribute specifies that the data item _atom_site_label.

1.7.1.3 DDL1 example 3

Here is a more complicated DDL1 definition for the six anisotropic atomic displacement parameters U^{ij}.

```
data_atom_site_aniso_U_
    loop_ _name                 '_atom_site_aniso_U_11'
                                '_atom_site_aniso_U_12'
                                '_atom_site_aniso_U_13'
                                '_atom_site_aniso_U_22'
                                '_atom_site_aniso_U_23'
                                '_atom_site_aniso_U_33'
    _category                    atom_site
    _type                        numb
    _type_conditions             esd
    _list                        yes
    _list_reference             '_atom_site_aniso_label'
    _related_item               '_atom_site_aniso_B_'
    _related_function            conversion
    _units                      A^2^
    _units_detail               'angstroms squared'
    _definition
;               These are the standard anisotropic atomic displacement
                components in angstroms squared which appear in the
                structure factor term:

                T = exp{-2pi^2^ sum~i~ [sum~j~ (U^ij^ h~i~ h~j~ a*~i~
                                                a*~j~) ] }

                h = the Miller indices
                a* = the reciprocal-space cell lengths

                The unique elements of the real symmetric matrix are
                entered by row.
;
```

Note the following aspects of this definition.

- the _related_item attribute identifies items that are related to the defined one. The nature of this relationship is specified with _related_function. In this case the value is conversion, which means that the U^{ij} can be derived directly from the B^{ij}.

- the _units attributes specify the units or dimensions of the U^{ij} in ångstroms squared.

1.7.2 DDL2 definition examples

The definitions shown above are from the CIF Core dictionary and illustrate how the DDL1 attributes are used to define data items. The DDL1 approach makes minimum use of the 'category' of data items, such as atom_site. In a sense this is inefficient because data attributes such as _list, _list_reference, _list_link_child, _list_link_parent, refer to properties of the class or category rather than to individual items. The DDL2 approach [Westbrook, Hall 96] uses a more hierarchical approach to data classes in which data items of a particular category are organized into a single table. The DDL2 also provides for explicit sub-categories in which data items are identified by function, e.g. 'matrix'. Although this approach is less intuitive to the casual user, it has proven to be advantageous in defining complex data relationships, such as those in the

macromolecular dictionary, and is therefore expected to be of increasing importance in the future as cross-discipline data bases develop.

1.7.2.1 DDL2 example 1

To illustrate the differences between the dictionary approaches, we shall now look at the _atom_site definitions in DDL2.

```
save__atom_site.fract_x

    _item_description.description
;               The x coordinate of the atom site position specified as a
                fraction of _cell.length_a.
;
    _item.name                    '_atom_site.fract_x'
    _item.category_id               atom_site
    _item.mandatory_code            no
    _item_aliases.alias_name      '_atom_site_fract_x'
    _item_aliases.dictionary        cif_core.dic
    _item_aliases.version           2.0.1
     loop_
    _item_dependent.dependent_name
                                  '_atom_site.fract_y'
                                  '_atom_site.fract_z'
    _item_related.related_name    '_atom_site.fract_x_esd'
    _item_related.function_code     associated_esd
    _item_sub_category.id           fractional_coordinate
    _item_type.code                 float
    _item_type_conditions.code      esd
     save_
```

Note the following aspects in this definition.

- DDL2 definitions are enclosed in a save frame, not a data block.
- DDL2 data names contain a dot '.' character that separates the category (starting the name) from the identity (ending the name).
- in DDL2 definitions each data item, independent of its irreducible relationship to other data items, is defined separately.
- DDL2 data names are equivalenced to other identical data items, including the DDL1 defined names, with the attributes _item_aliases.alias_name values.
- in DDL2 the _item_type.code attribute is identical to the DDL1 _type attribute, except that it has a more detailed enumeration e.g. *number* has been expanded to *integer*, *float*, etc.

1.7.2.2 DDL2 example 2

Here is another DDL2 definition to emphasise the differences in definition approach.

```
save__atom_site.aniso_U[1][3]_esd

    _item_description.description
;               The estimated standard deviation of
                _atom_site.aniso_U[1][3].
;
    _item.name                    '_atom_site.aniso_U[1][3]_esd'
    _item.category_id               atom_site
    _item.mandatory_code            no
    _item_default.value             0.0
     loop_
    _item_related.related_name
    _item_related.function_code   '_atom_site.aniso_U[1][3]'
                                    associated_value
                                  '_atom_site.aniso_B[1][3]_esd'
                                    conversion_constant
                                  '_atom_site_anisotrop.B[1][3]_esd'
                                    conversion_constant
                                  '_atom_site.aniso_B[1][3]_esd'
                                    alternate_exclusive
                                  '_atom_site_anisotrop.B[1][3]_esd'
                                    alternate_exclusive
                                  '_atom_site_anisotrop.U[1][3]_esd'
                                    alternate_exclusive
    _item_sub_category.id           matrix
    _item_type.code                 float
    _item_units.code                angstroms_squared
     save_
```

Note the following in this definition.

- DDL2 defines the esd (or su) of U13 as a separate data item, whereas in DDL1 the su is assumed to be appended to the value.
- an additional classification attribute `_item_sub_category.id` is defined in DDL2.
- in DDL2 definitions each data item, independent of its irreducible relationship to other data items, is defined separately.

1.7.2.3 DDL2 example 3

Finally here is how the properties of the *category* `atom_site` are defined in DDL2.

```
save_ATOM_SITE
    _category.description
;               Data items in the ATOM_SITE category record details about
                the atom sites in a macromolecular crystal structure,
                such as the positional coordinates, atomic displacement
                parameters, magnetic moments and directions, and so on.

                The data items for describing anisotropic temperature or
                thermal displacement factors are only used if the
                corresponding items are not given in the
                ATOM_SITE_ANISOTROP category.
;
    _category.id                    atom_site
    _category.mandatory_code        no
    _category_key.name             '_atom_site.id'
     loop_
    _category_group.id              'inclusive_group'
                                    'atom_group'
     save_
```

Note the following aspects in this category definition.

- in DDL2 the attribute _category_key.name, which is equivalent to the DDL1 _list_reference, is defined only once, whereas in DDL1 must be declared in the definition of each data item.

- the category attributes are identified by the name structure _category_ as opposed to the _item_ prefix used to define data items. It is important to emphasise that atom_site is NOT a data item and will not appear in a CIF.

1.8 Handling DDL1 and DDL2 name structures

The different naming structures in the two dictionary languages, DDL1 and DDL2, appears to complicate the use of CIFs. This is avoided because the *CIFtbx* toolbox handles these naming convention transparently and interchangeably provided there is access to the relevant dictionaries.

We shall now look quickly at some data items expressed in both conventions. Here is an extract of a CIF containing core data items.

```
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
    _atom_site_calc_attached_atom
      O1   .4154(4)  .5699(1)  .3026(0)  .060(1)  Uani  ?    ?
      C2   .5630(5)  .5087(2)  .3246(1)  .060(2)  Uani  ?    ?
      C3   .5350(5)  .4920(2)  .3997(1)  .048(1)  Uani  ?    ?
      N4   .3570(3)  .5558(1)  .4167(0)  .039(1)  Uani  ?    ?
      C5   .3000(5)  .6122(2)  .3581(1)  .045(1)  Uani  ?    ?

loop_
    _atom_site_aniso_label
    _atom_site_aniso_U_11
    _atom_site_aniso_U_22
    _atom_site_aniso_U_33
    _atom_site_aniso_U_12
    _atom_site_aniso_U_13
    _atom_site_aniso_U_23
    _atom_site_aniso_type_symbol
     O1   .071(1) .076(1) .0342(9) .008(1)   .0051(9) -.0030(9) O
     C2   .060(2) .072(2) .047(1)  .002(2)   .013(1)  -.009(1)  C
     C3   .038(1) .060(2) .044(1)  .007(1)   .001(1)  -.005(1)  C
     N4   .037(1) .048(1) .0325(9) .0025(9)  .0011(9) -.0011(9) N
     C5   .043(1) .060(1) .032(1)  .001(1)  -.001(1)   .001(1)  C
```

In this CIF, the anisotropic atomic displacement parameters have been looped in a separate list from the atomic coordinates. The each row in the second list is linked to a row in the list of atomic coordinates by the value of `_atom_site_aniso_label` that matches the value of `_atom_site_label` in the associated row of the list of atomic coordinates. Though it is customary to align the ordering of the two lists, CIF does not require them to be in the same order, only that the labels can be matched. Alternatively, the two lists could have been merged into one, using the same tags.

In the DDL2-based *mmCIF* dictionary there are two alternate sets of names for presentation of anisotropic atomic displacement parameters, one set in the `atom_site` category, and another set in a distinct `atom_site_anisotrop` subcategory. In a CIF one set of names can be used but not both. If the names from the parent category are used they must be combined with these items. An atomic coordinate list with anisotropic displacement parameters merged into the same list in mmCIF would look like this.

```
 loop_
_atom_site.label_seq_id
_atom_site.auth_asym_id
_atom_site.group_PDB
_atom_site.type_symbol
_atom_site.label_atom_id
_atom_site.label_comp_id
_atom_site.label_asym_id
_atom_site.auth_seq_id
_atom_site.label_alt_id
_atom_site.cartn_x
_atom_site.cartn_y
_atom_site.cartn_z
_atom_site.occupancy
_atom_site.B_iso_or_equiv
_atom_site.footnote_id
_atom_site.label_entity_id
_atom_site.id
_atom_site.aniso_U[1][1]
_atom_site.aniso_U[1][2]
_atom_site.aniso_U[1][3]
_atom_site.aniso_U[2][2]
_atom_site.aniso_U[2][3]
_atom_site.aniso_U[3][3]

1 . ATOM  N  N  GLU * 1  A   4.127  26.179  -7.903  0.49 57.53  .  1   1
                         0.9336  0.0004  0.2737  0.7394  0.2771  0.4591
1 . ATOM  N  N  GLU * 1  B   3.535  25.488 -12.889  0.51 54.52  .  1   2
                         0.8406 -0.0887  0.3093  0.5015  0.0161  0.6783
1 . ATOM  C  CA GLU * 1  A   5.490  26.607  -8.207  0.49 52.50  .  1   3
                         0.9283 -0.0256  0.2331  0.5563  0.1241  0.4611
1 . ATOM  C  CA GLU * 1  B   2.754  26.395 -12.051  0.51 51.27  .  1   4
                         0.7663 -0.0653  0.2258  0.5124  0.0184  0.6212
1 . ATOM  C  C  GLU * 1  A   5.550  27.734  -9.233  0.49 47.55  .  1   5
                         0.8593  -0.088   0.182  0.4752  0.0625  0.4275
```

The same information presented as two lists in *mmCIF* would use different tags, very similar to those used in DDL1.

```
 loop_
_atom_site.label_seq_id
_atom_site.auth_asym_id
_atom_site.group_PDB
_atom_site.type_symbol
_atom_site.label_atom_id
_atom_site.label_comp_id
_atom_site.label_asym_id
_atom_site.auth_seq_id
_atom_site.label_alt_id
_atom_site.cartn_x
_atom_site.cartn_y
_atom_site.cartn_z
_atom_site.occupancy
_atom_site.B_iso_or_equiv
_atom_site.footnote_id
_atom_site.label_entity_id
_atom_site.id
_atom_site.aniso_U[1][1]
_atom_site.aniso_U[1][2]
_atom_site.aniso_U[1][3]
_atom_site.aniso_U[2][2]
_atom_site.aniso_U[2][3]
_atom_site.aniso_U[3][3]

1 . ATOM  N  N  GLU * 1  A   4.127  26.179  -7.903  0.49 57.53  .  1   1
1 . ATOM  N  N  GLU * 1  B   3.535  25.488 -12.889  0.51 54.52  .  1   2
1 . ATOM  C  CA GLU * 1  A   5.490  26.607  -8.207  0.49 52.50  .  1   3
1 . ATOM  C  CA GLU * 1  B   2.754  26.395 -12.051  0.51 51.27  .  1   4
1 . ATOM  C  C  GLU * 1  A   5.550  27.734  -9.233  0.49 47.55  .  1   5

_atom_site_aniso.id
_atom_site_aniso.U[1][1]
_atom_site_aniso.U[1][2]
_atom_site_aniso.U[1][3]
_atom_site_aniso.U[2][2]
_atom_site_aniso.U[2][3]
_atom_site_aniso.U[3][3]

1                      0.9336  0.0004  0.2737  0.7394  0.2771  0.4591
2                      0.8406 -0.0887  0.3093  0.5015  0.0161  0.6783
3                      0.9283 -0.0256  0.2331  0.5563  0.1241  0.4611
4                      0.7663 -0.0653  0.2258  0.5124  0.0184  0.6212
5                      0.8593  -0.088   0.182  0.4752  0.0625  0.4275
```

The *mmCIF* the `_atom_site_aniso.id` tag fills the role of the small molecular core CIF tag `_atom_site_aniso_label` for which it is an alias, and the mmCIF tag `_atom_site.id` tag fills the role of the small molecule core CIF tag `_atom_site_label` for which it is an alias, providing the same approach to linking the list of anisotropic atomic displacement parameters to the list of atomic coordinates.

CHAPTER 2

Overview of the Tool Box

2.1. Introduction

The *CIFtbx* library is made up of Fortran functions, subroutines, monitor variables and control variables . It is used to develop software to read and / or write CIF data. In addition, these software "tools" automatically test the validity of incoming CIF data, and ensure the correct deposition of outgoing data. The self-checking aspects of these tools are important for ensuring that the data structure of the CIF is correct, and, when used in association with the DDL dictionaries, that the individual items and lists are conformant to the data definitions.

This chapter provides an overview of the available *CIFtbx* tools. The next chapter shows how these tools are applied to different data manipulation tasks, and later chapters provide the reference manual details on how the various options for tools are invoked and interpreted.

CIFtbx facilities are of four types:

1. commands to **initialise** later handling,
2. commands to **read** CIF data,
3. commands to **write** CIF data,
4. variables for **monitor** and **control** signals.

CIFtbx **commands** are Fortran function or subroutine calls which are invoked in the standard way. For example, to open the dictionary file "`core.dic`" one would simply enter the logical function `dict_`

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

FN is a local *LOGICAL* variable. The string `'core.dic'` is the local file identifier for the dictionary. The string `'valid'` informs the command of the checking that should be done against the data definitions in this dictionary. If `dict_` opens the file "`core.dic`" correctly the value of FN is returned as `.true.`; otherwise the function is returned as `.false.`

If the command is a subroutine, such as `purge_`, which is used to clear the internal data tables, it is invoked as

```
call purge_
```

Note that all but two of the commands are functions.

The arguments to the commands are minimal. Much of the work of the library is done by reading and setting variables held in common blocks. Both the common block declarations and the type declarations for all the commands are provided in the file `ciftbx.cmn`, which must be 'included' in each program, function or subroutine that uses *CIFtbx*.

The flexibility of CIF presents some challenges to the writer of applications that use CIF. The information in a CIF may be presented in any order, with the data names presented in upper or lower case and with whatever spacing between items pleases one's taste. You may take a CIF, pick up, say, the cell parameters from the front of the file, place them at the end, after all the atomic coordinates, change the "`_cell...`" in all the data names to "`_Cell...`" and introduce a blank line between each data name and its value and call the file the same CIF. *CIFtbx* provides the application writer with the tools to process both the original file and the modified file as the same CIF. When the application needs the cell parameters, it asks for them by name. For the advanced application writer, *CIFtbx* also has the option of simply asking for the next data item, whatever the name of that data item might be, allowing the application to go beyond the position-independent context of CIF and to be sensitive to the position of items within the CIF.

2.2. Initialisation Commands

Initialisation commands are applied before any other commands. There are only two tools in this category.

```
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 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 5-character 'check codes', such as `'valid'` or `'dtype'` to turn on checking for the validity of tags or types of values. `dict_` is a logical function which is returned as `.true.` if the name dictionary was opened and if the check codes are recognisable.

2.3. Read Commands

These commands are used to read data from an existing CIF. Since CIF data is order-independent, most applications would work from a known list of data names (tags) and to 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_ A bookmark command saves or restores the current position in the CIF so that data can be accessed non-sequentially, 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 deviation, `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 deviation, `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 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.

2.4. 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 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 ploop_ (name)
     character name*(*)
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 "data_*name*", 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 "save_*name*" instead of a data block 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. 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 data names, the first character of which is blank, or the loop_ will be terminated.

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 esd, 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 esd, 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 ";" 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_ (default 80 characters). This function is useful when embedding a CIF 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.

2.5. Variables

The *CIFtbx* library also contains a large number of variables declared in the common blocks in the file ciftbx.cmn that may be used to monitor details of reading and writing processes and to control various functions. 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. See Chapter 7 for more information.

2.5.1. General Monitor Variables

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

file_ *Character string* containing the filename of the current input file.

longf_ *Integer variable* containing the length of the filename 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*: which is the *CIFtbx* version and date in the form 'CIFtbx version N.N.N DD MMM YYYY '[1]

[1]The formatof this string changed in version 2.6 to become year-2000 compliant. In prior versions this string was 'CIFTBX version N.N.N, DD MMM YY'.

2.5.2. General Control Variables

These variables are *specified* to control *CIFtbx* commands.

alias_ *Logical variable* to control the use of data name aliases for input items. If set `.true.` aliases from the input dictionary may be used (see 2.6 below). The default is `.true.`

append_ *Logical variable*: to control reuse of the direct access file. If set `.true.` 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* controls the treatment of input blank strings. If set `.true.` `char_` or `test_` will return the type as *null* rather than *char* when encountering a quoted blank.

recbeg_ *Integer variable* that gives the record number of the first record to be used. May be changed by the user to restrict access to a CIF.

regend_ *Integer variable* that gives 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.`

2.5.3. Input Monitor Variables

These variables are *returned* by *CIFtbx* tools and are used to decide on subsequent actions in the program. Note that 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 there is 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 2.6, below) 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_`.

esddig_ *Integer variable* containing number of esd digits in the last number read from a CIF. Will be zero if no esd was given.

glob_ *Logical variable*: is `.true.` if the current data block is actually 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 which are not in a list (a zero value).

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

long_ *Integer variable* containing the length of the data string in `strg_`.

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.

uote_ *Character variable* giving the quotation symbol found delimiting the last string read.

save_ *Logical variable* is `.true.` if the current data block is actually 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 4-character strings, `'null'` (for missing data, the period or the question mark), `'numb'` (for numeric

data), 'char'(for most character data) or 'text' (for semi-colon delimited multi-line character data)[2].

2.5.2. Output Control Variables

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

aliaso_ *Logical variable* to control the use of data name aliases for output items. If set .true. preferred synonyms from the input dictionary may be output (see 2.6 below). 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 esd integers output by pnumb_. The default value is 19, which limits esd's to the range 2-19.

globo_ *Logical variable* which if set .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 .true. output quoted blank strings will be converted to an unquoted period (i.e. to a data item of type *null*)[3].

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

pesddig_ *Integer variable* to set non-zero, and esdlim_ is negative, controls the number of digits for esd's produced by pnumb_ and pnumd_

plzero_ *Logical variable* controls the treatment of leading zeros in output decimal numbers. If set .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.

---

[2]For most purposes the type text is a sub-type of the type char, and not a distinct data type. *CIFtbx* permits multi-line text fields to be used whenever character strings are expected.
[3]*CIFtbx* treats an unquoted period or question mark as being of type 'null'.

pposval_ *Integer variable* to set 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 `.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.`

2.6. 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 between DDL1 and DDL2 dictionaries (this was discussed in Chapter 1), 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 may 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 normally 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 variables `alias_` and `aliaso_`. When `alias_` is set `.false.` the automatic recognition and translation of aliases stops. When `aliaso_` is set `.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 `.false.`, `dicname_` will agree with 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_aniso_U_11` from the core dictionary is the alias of `_atom_site_anisotrop.u[1][1]` in the mmCIF dictionary. In the following application of *CIFtbx* function `test_` the specified data name `_atom_site_aniso_U_11` is used to inquire as to the names used in an input CIF.

```
      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_
```

Invocation 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
```

CHAPTER 3

How to Use the Tool Box

3.1. Introduction

The *CIFtbx* tool box is supplied as a suite of Fortran source files and test files. The installation instructions for *CIFtbx* are given in Appendix B.

In this chapter we will provide an overview on how to use the *CIFtbx* tools. This will be done using a series of simple application examples. These examples are similar to that supplied with the test software.

The main source files[4] of *CIFtbx* are:

`ciftbx.f`	
`ciftbx.sys`	(used in `ciftbx.f`)
`ciftbx.cmn`	(used in local applications)

These may be applied to local software in several different ways:

1. Compile the `ciftbx.f` and link the resulting object file either as an object *library*, or explicit references[5] in the linking sequence.

2. Include `ciftbx.f` in the local software code, so that the toolbox will be compiled and linked as part of the local development.

Clearly approach 1. is more efficient because the toolbox is only complied once. However, for some small applications approach 2. may be simpler.

Note that the common file `ciftbx.cmn` must be 'included' into any local routines that use *CIFtbx* tools. This will be illustrated in the later examples.

[4]Versions 2.6 and earlier of CIFtbx require certain additional source files: ciftbx.cmf, ciftbx.cmv, hash_funcs.f and clearfp.f (or clearfp_sun.f)
[5]Versions 2.6 and earlier of CIFtbx require hash_funcs.o as an additional object file.

3.2. Reading CIF data

The *CIFtbx* approach to reading a CIF is best illustrated with a simple example. A source code listing of the program CIF_IN is shown below. This program reads the file `test.cif` (shown after the source listing). While it is doing this it tests the input data items against the dictionary file `cif_core.dic`. The output from running CIF_IN is shown below.

```
          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
```

The logic of the program CIF_IN is basically as follows:

A Define the variables used in the program. The common variables for *CIFtbx* functions are added with the line '`include ciftbx.cmn`'.

B Assign specific device numbers to the various files used in the program 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 the dictionary file '`cif_core.dic`' with the command **`dict_`**. This also specifies with the code 'valid' that the input data items be validated against the dictionary. Note that `dict_` is invoked inside an IF statement which tests if it is `.true.` and therefore successful.

D Open the CIF '`test.cif`' with the command **`ocif_`** and test the returned logical value to see if the file is opened.

E Use the **`data_`** command, containing a *blank* block code, to initiate subsequent data entries at the *next* encountered data block. The data block name is returned in the variable **`bloc_`** which is printed.

F Read the cell length values, and their standard deviations, with **`numb_`** and print these out. Test that 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 identify the next encountered data item.

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 if 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_`**.

Here is a listing of the input CIF '`test.cif`'.

```
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
```

Executing CIF_IN produces the following printout.

```
 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
```

Note the following aspects of the CIF_IN approach to reading data.

- The first two lines of the printout (in red) were generated by the *CIFtbx* library routines, not by the program CIF_IN. These messages result from the checking of the input CIF against the dictionary '`cif_core.dic`', as requested by the `dict_` command. Note also that dictionary validation messages are issued on the execution of the `data_` command, because this is when all data items in the designated data block (in this case `mumbo_jumbo`) are read from the CIF, stored in a scratch file, checked against the dictionary and pointers all pointers and attributes of the items are recorded. All subsequent commands use these pointers and the scratch file to access the data.

- The '`####`' string in front of `_cell_length_a` makes this data item a comment and CIF_IN detects it, via the logical variable `f1`, as missing.

- Items may be read from the CIF in any order, with the exception that items in the same looped list should be accessed together. If you need to access items in different lists simultaneously, then the `bkmrk_` command must be used to prevent the *CIFtbx* loop pointers from being mistakenly reset.

3.3. Reading text data in loops

This example illustrates how text data is read from lists. This is a typical requirement when reading address labels or audit trails from a CIF.

Here is the file `paper.cif` that needs to be read.

```
data_publication

loop_
_publ_author_name
_publ_author_address
  'Furber, Mark'
;
Research School of Chemistry
Australian National University
GPO Box 4
Canberra, A.C.T.
Australia    2601
;
  'Mander, Lewis N.'
;
Research School of Chemistry
Australian National University
GPO Box 4
Canberra, A.C.T.
Australia    2601
;
```

Here is an *extract* from a routine that reads the above addresses.

```
         if(data_('publication'))
     *   write(6,'(a,a/)') ' Access items in data block  ',bloc_
C
         write(6,'(a)') ' Author list'
210      f1 = char_('_publ_author_name', line)
         write(6,'(/1x,a)') line(1:long_)
220      f1 = char_('_publ_author_address', line)
         if(line(1:10).eq.'          ') goto 230
         write(6,'(1x,a)') line(1:50)
230      if(text_) goto 220
         if(loop_) goto 210
```

The relevant printout from running this routine follows.

```
 Access items in data block  publication

 Author list

 Furber, Mark
 Research School of Chemistry
 Australian National University
 GPO Box 4
 Canberra, A.C.T.
 Australia    2601

 Mander, Lewis N.
 Research School of Chemistry
 Australian National University
 GPO Box 4
 Canberra, A.C.T.
 Australia    2601
```

Note the following aspects of this run.

- The `char_` command is used to read both the character string item `_publ_author_name` and the text lines `_publ_author_address`.

- The `text_` variable is used to monitor whether another text line is present in the CIF, and the `loop_` variable is used to monitor if there is another name/address packet is present in the loop.

3.4. Reading user-requested data items

The first two examples were used to show how data items are read from a CIF when the required data names were known in advance, i.e. when pre-ordained data items needed to be accessed. In such applications the data names can be 'hardwired' into the program code. What about those applications where the input data items are determined by user requests? The lack of advance knowledge on what items are to read leads to important differences in the programming approach, because now the *data attributes* of these items cannot be assumed. Either these may be determined from the nature of the input *values*, or from the dictionary.

The next program example illustrates how a general list of data requested from an input CIF may be handled. The logical function `test_` is used to identify the data *type* of the requested data item, and then the appropriate `numb_` or `char_` is applied to enter the data value. Note that the list of requests used in the file '`test.req`' is not of particular significance for this example; they have been intentionally jumbled with respect to the input CIF '`test.cif`' (see 3.2) to show what happens if a non-loop item is accessed within a loop sequence. The *CIFtbx* routines treat such a "rogue" request as a signal to terminate the loop sequence so that the next call for a loop item will restart the sequence and extract data from its first packet!

Here is an extract of some code that enters the input requests from '`test.req`' (shown below), and the print the items, their attributes and their values. The printout is shown last.

```
      open(unit=8,file='test.req',status='old')

300   read(8,'(a)',end=400) name
      if(.not.test_(name))     goto 300
C
      if(type_.ne.'numb')      goto 320
      f1 = numb_(name, numb, sdev)
      write(6,'(a,3x,a,2i5,2f10.4)') name,type_,long_,list_,numb,sdev
      goto 300
320   if(type_.ne.'char')      goto 340
      f1 = char_(name, line)
      write(6,'(a,3x,a,2i5,a)') name,type_,long_,list_,line(1:long_)
      goto 300
340   if(type_.ne.'text')      goto 300
      write(6,'(a,3x,a,2i5)') name,type_,long_,list_
350   f1 = char_(name, line)
      write(6,'(a)') line
      if(text_)  goto 350
      goto 300
```

Here is the list of data names in the request file '`test.req`'.

```

_dummy_test
_audit_creation_date
_audit_creation_method
_audit_update_record
_chemical_name_systematic
_chemical_formula_moiety
_chemical_formula_weight
_chemical_melting_point
_cell_length_a
_cell_length_b
_cell_length_c
_cell_measurement_theta_min
_cell_measurement_theta_max

_blat2
_blat1
_blat2
_blat1
_blat2
_blat1
_blat2
_blat1
_blat2
_blat1

_symmetry_cell_setting
```

The printout for this example run follows.

```
_dummy_test                  char   30  0 rubbish to see what dict_ says
_audit_creation_date         char    8  0 91-03-20
_audit_creation_method       char   34  0
from_xtal_archive_file_using_CIFIO
_audit_update_record         text   56  0
  91-04-09           text and data added by Tony Willis.
  91-04-15           rec'd by co-editor with diagram as manuscript HL7.
_chemical_formula_moiety     char   12  0 C18 H25 N O3
_chemical_formula_weight     numb    6  0  303.4000    0.0000
_chemical_melting_point      null    1  0
_cell_length_b               numb    9  0  14.9560    0.0010
_cell_length_c               numb    9  0  19.7370    0.0030
_cell_measurement_theta_min  numb    2  0  25.0000    0.3000
_cell_measurement_theta_max  numb    2  0  31.0000    0.3000
_blat2                       char    1  2  2
_blat1                       char    1  2  1
_blat2                       char    1  2  4
_blat1                       char    1  2  3
_blat2                       char    1  2  6
_blat1                       char    1  2  5
_blat2                       char    1  2  b
_blat1                       char    1  2  a
_blat2                       char    1  2  d
_blat1                       char    1  2  c
_symmetry_cell_setting       char   12  0  orthorhombic
```

Note the following for this example.

- The command `test_` is returned `.true.` if the specified data name (in this case input from 'test.req') is present in the input CIF. If it is not the test example simply reads another name.

- The `test_` command sets the variables `type_`, `long_` and `list_` and these are printed out. The possible values for `type_` are *null*, *char* and *numb* (a DDL1 dictionary is used in this case). The variable `long_` is the length of the value string and `list_` is the sequential number of the list block.

- The value **?** for `_chemical_melting_point` is treated as a *null* string of length 1.

- The `_cell_length_a` item is treated as missing because of the preceding hashes so that the value of `test_` is .false. and is skipped.

- The requests for `_blat2` and `_blat1` are interpreted according to the values present in the list. Note that this is not intended to be a sensible list of data and mixes numbers and character strings. It is shown here simply to illustrate how the value of `type_` is returned. For these data items the value of `list_` is 2 because they reside in the second looped list of the data block `mumbo_jumbo`.

- Note that the request for _symmetry_cell_setting terminates the loop block settings, and an further requests for _blat2 and _blat1 would start at the first packet of this block.

3.5. Creating a CIF

We will now show how to generate CIF data items. The following example program creates a CIF. For the sake of clarity the source code, and the generated CIF 'test.new' (shown later), are kept very simple. The initial data definition part of this program has also been omitted for brevity.

```
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
C....... Request dictionary validation check
450      if(dict_('cif_core.dic','valid'))  goto 260
         write(6,'(/a/)') ' Requested Core dictionary not present'
C
C....... Insert a data block code
460      f1 = pdata_('whoops_a_daisy')
C
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
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
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!')
C
```

```
C....... Now output some comments and various numeric and esd formats
         f1 = pcmnt_(' ')
         f1 = pcmnt_(' Loops with various numeric and esd formats')
         f1 = ploop_('_numeric_data_1')
         f1 = ploop_('_numeric_data_2')
         f1 = ploop_('_numeric_data_3')
         f1 = ploop_('_numeric_data_4')
         esdlim_ = 19
         pdecp_ = .false.
         plzero_ = .false.
         f1 = pcmnt_(' ')
         f1 = pcmnt_(' esdlim_=19, pdecp_=.false., plzero_=.false.')
         f1 = pnumb_(' ', -.01, 1.)
         f1 = pnumb_(' ', -.1, 10.)
         f1 = pnumb_(' ',-1.,100.)
         f1 = pnumb_(' ',1.,100.)
         pdecp_ = .true.
         plzero_ = .false.
         f1 = pcmnt_(' ')
         f1 = pcmnt_(' esdlim_=19, pdecp_=.true., plzero_=.false.')
         f1 = pnumb_(' ', -.01, 1.)
         f1 = pnumb_(' ', -.1, 10.)
         f1 = pnumb_(' ',-1.,100.)
         f1 = pnumb_(' ',1.,100.)
         esdlim_ = -9999
         plzero_ = .true.
         f1 = pcmnt_(' ')
         f1 = pcmnt_(' esdlim_=-9999, pdecp_=.true., plzero_=.true.')
         f1 = pnumb_(' ', -.01, 1.)
         f1 = pnumb_(' ', -.1, 10.)
         f1 = pnumb_(' ',-1.,100.)
         f1 = pnumb_(' ',1.,100.)
500      call close_
         stop
         end
```

Here is the contents of the created CIF '`test.new`'.

```
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!
;

# Loops with various numeric and esd formats
loop_
    _numeric_data_1             #< not in dictionary
    _numeric_data_2             #< not in dictionary
    _numeric_data_3             #< not in dictionary
    _numeric_data_4             #< not in dictionary

# esdlim_=19, pdecp_=.false., plzero_=.false.
       -.0(10)     -0(10)    -0E1(10)      0E1(10)

# esdlim_=19, pdecp_=.true., plzero_=.false.
       -.0(10)    -0.(10)   -0.E1(10)     0.E1(10)

# esdlim_=-9999, pdecp_=.true., plzero_=.true.
    -0.01(100)  -0.1(100)    -1.(100)      1.(100)
```

Note the following aspects of this approach.

- Because the dictionary command `dict_` was used, each data name output is checked against the dictionary '`cif_core.dic`'. Unrecognised names are flagged with the comment '`#< not in dictionary`'. This applies to both looped and single data items.

- Note in the list of `_numeric_data_` values how the number format control variables `esdlim_`, `esdlim_` and `plzero_` differ.

3.6. General tips on applying *CIFtbx*.

3.6.1. Reading a CIF

The basic steps for reading a CIF are:

- Load any dictionaries required for checking
- Open the CIF to be read
- Locate the desired `data_` block within the CIF
- Enter the required data items by name
- Note that only one input CIF may be open

- Note that only one `data_` block may be accessed
- Note that within a `data_` block, only one `loop_` may be accessed unless the bookmark command `bkmrk_` is used to keep the place in multiple loops

3.6.2. Writing a CIF

For applications writing CIF data, the basic steps needed are

- Load dictionaries required for checking
- Open the output CIF by name
- Write the `data_` block header
- Write the tags and associated data values for non-`loop_` items and/or write a `loop_` header followed by all its associated values
- Repeat this process for additional `data_` blocks
- Close the CIF
- Note that only one CIF may be written at once, but an input CIF may also be open
- Note that only one `data_` block may be written at once
- Note that within a `data_` block, only one `loop_` may be written at once

3.6.3. Program organisation

The general structure of an application program using the *CIFtbx* tools is

- Declarations needed by the application
- Insert `"include 'ciftbx.cmn'"`
- Initialize *CIFtbx* variables
- Use *CIFtbx* initialization and dictionary loading commands
- Use functions to open input CIF and/or output CIF
- Use functions to read and/or write data items
- Calls to subroutine to close CIF

CHAPTER 4

Reference: Initialisation Functions

4.1. Introduction

CIFtbx provides two initialisation functions, `init_` and `dict_`, which affect all other CIF reading and writing operations. The `init_` function is used to assign unit (device) numbers for files used in the *CIFtbx* processes. The `dict_` function inputs the data dictionaries and specifies the conditions by which input and output data items are checked.

These functions, if used, must precede all other *CIFtbx* commands. If the `init_` command is used to change the default unit number for a dictionary file, it must precede the relevant `dict_` command. The command `dict_` is repeated for as many dictionaries as need to be attached to the application.

4.2. `init_`

```
logical function init_ (devcif, devout, devdir, deverr)
          integer devcif, devout, devdir, deverr
```

4.2.1 Definition

`init_` is an *optional logical* function for specifying the Fortran device numbers for the four *CIFtbx* files: the input CIF, the output CIF, an internal scratch file and the error message file. The value of the logical function is always returned as .true.

The `init_` arguments are:

No.	*Augment*	*Description*	*Default value*
1	<input CIF dev number>	input CIF device	1
2	<output CIF dev number>	output CIF device	2
3	<scratch file dev number>	scratch file device	3
4	<error file dev number>	error message device	6 (*stdout*)

Note that dictionary files are read using the device number specified by argument 1.

4.2.2 Application

`init_` is a logical function that can be applied in a variety of ways. The typical coding to apply this function is

```
logical flag
....
flag = init_(arg1,arg2,arg3,arg4)
```

The value returned in `flag` will always be .true. The device numbers are used by *CIFtbx* to open files, but the opening is *not* initiated by the `init_` call. Note that in some circumstances it may be necessary to invoke this command more than once in an application. For example,

```
flag = init_(1,6,3,0)
flag = dict_('cif_mm.dic','valid dtype')
flag = init_(5,6,3,0)
```

is used to attach the dictionary file as unit 1 and after this has been loaded, change the input CIF unit to 5.

Optimal device numbers

The choice of device numbers is application and system dependent. For Unix systems, it is a good practice to use device number 0 for the error device number (*arg4*), where unit 0 is the standard error device *stderr*. For most OS's, device number 6 is the default print device, and is a good choice for the error device number (*arg4*), if 0 is not available.

For many systems, device numbers 5 and 6 have special meaning as the default input *stdin* and list devices *stdout*. If no dictionaries are to be loaded, then 5 may be appropriate for the input CIF device number (*arg1*) and 6 appropriate for the output CIF device number (*arg2*). If a dictionary needs to be loaded, or if the input CIF needs to be a non-default file, then the use of unit 1 should avoid conflicts with default system behaviour.

If the local OS enforces carriage-control on unit 6, or the output CIF needs to be written to a non-default file, then unit 2 should avoid conflicts. When the input CIF (*arg1*) is opened (with `ocif_`) or a dictionary is opened (with `dict_`), the data is copied to the direct-access file (*arg3*), and then closed. No close is done if the Fortran `open` was suppressed (see `ocif_`) . Because *CIFtbx* immediately copies an input CIF to the scratch file, it is acceptable for an input CIF to be on a read-once device, such as the standard input device (*stdin=5*) on Unix systems.

4.3. `dict_`

```
logical function dict_ (fname, checks)
          character fname*(*), checks*(*)
```

4.2.2 Definition

`dict_` is an *optional logical* function used to read a CIF dictionary file (or a file containing a list of dictionary filenames) and to initiate optional data checks. The

logical function is returned as .true. if the named dictionary was opened, and if the check codes are recognisable.

The command arguments are:

No.	*Augment*	*Description*	*Default value*
1	<dictionary file name>	filename of a dictionary, *or* of a file† containing a list of dictionary filenames, *or* blank if only the checking codes are to be changed. († the format of this file is described below)	
2	<checking codes>	codes use to initiate CIF checks are entered as a single quoted string separated by blanks: 'valid' data name validation check 'catck' check data categories (default) 'catno' reverse of catck 'dtype' data type_ check 'reset' switch off all checking flags 'close' close existing dictionaries 'first' give first dictionary priority (default) 'final' give last dictionary priority 'nodup' forbid duplicate data names	

4.3.2 Application

dict_ is a logical function and can be applied in a variety of ways. Typically the required Fortran coding for the application of this function is

```
logical flag
....
flag = dict_(arg1,arg2)
if( dict_(arg1,arg2)) goto 100
```

The value returned in flag will be .true. only if the named dictionary file is opened *and* the listed check codes are recognised.

The filename in *arg1* must be appropriate for use in a standard Fortran `open` statement. For example, a dictionary is in a local directory could be `'cif_core.dic'`, whereas a dictionary file in a publicly accessible area might be `'/bin/public/cif_core.dic'` The sequence of checking codes in *arg2* must be enclosed in quotes and separated by blanks, e.g. '`valid dtype`'.

If the `dict_` command is used, it MUST be BEFORE the `data_` command, to which the data checks apply, is invoked.

Programmers (and users) needs to appreciate that when data item names are loaded from more than one dictionary, the names loaded first have priority over the same or aliased names loaded from subsequent dictionaries, *unless* the checking code `final` is specified, in which case the order of priority is reversed.

The `dict_` command sets the *CIFtbx* variable `dicname_` to either the specified filename of the opened dictionary, *or*, if the opened dictionary file contains the data item `_dictionary_name`, to its value. The variable `dicver_` is set to the value of `_dictionary_version`.

CHAPTER 5

Reference: Read Functions

5.1. Introduction

This chapter describes in detail the *CIFtbx* functions that read and check CIF data items. Descriptions in this chapter assume knowledge of general principles of CIF implementation and application covered in Chapter 3.

5.1.1 *CIFtbx* bookkeeping

As summarised in 3.6.1, there is a required sequence to some *CIFtbx* commands. Device numbers need to be specified, dictionaries loaded and the CIF to be read, opened. The command `data_` copies the data in the named data block to the direct access file and establishes a list of pointers that locate each data name (tag) in the block. This is shown diagrammatically in Fig. 5.1.

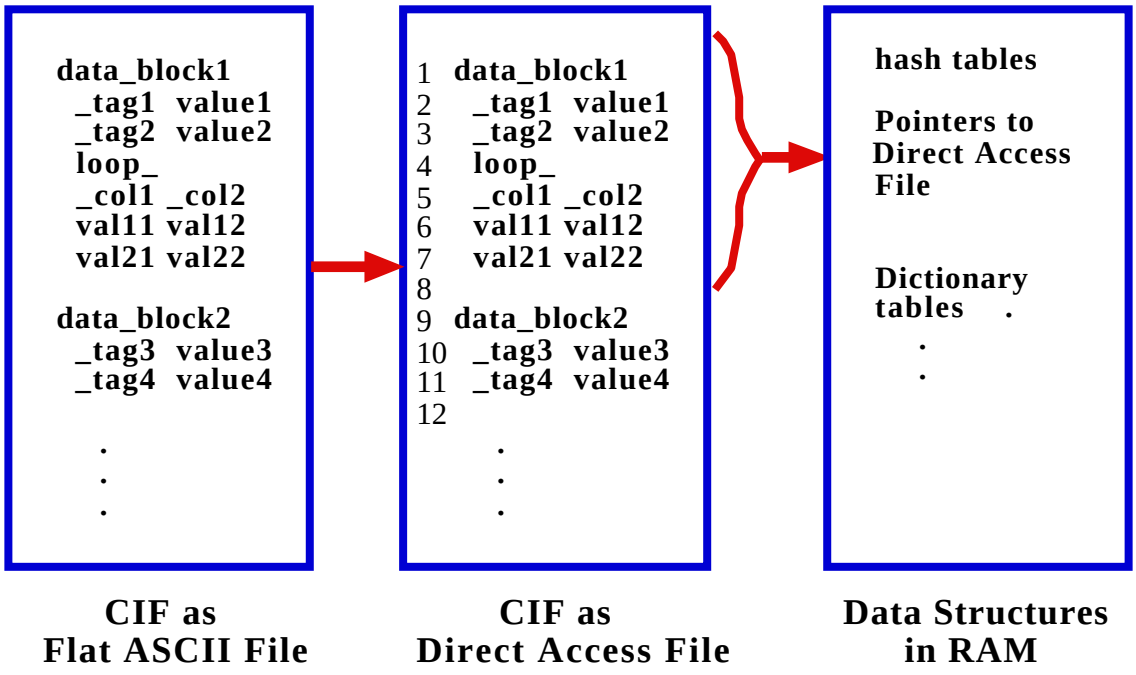

Fig. 5.1 Preparing to Read a CIF with CIFtbx

Normally only the data from one input CIF is stored in the direct access file. However, if more than one CIF needs to be processed simultaneously (e.g. there is a need to move between data blocks in different CIFs, the variable `append_` set to .true. retains previous CIFs in the direct access file.

5.2. ocif_

```
logical function ocif_ (fname)
     character fname*(*)
```

5.2.1 Definition

ocif_ is an *optional logical* function for opening an input CIF. The value of the function is returned as .true. if the named file has been opened.

The command argument is:

No.	*Argument*	*Description*
1	<input CIF filename>	input CIF filename string *in quotes*

5.2.2 Application

ocif_ can be applied in a variety of ways. Typically the coding of this function is

```
logical flag
....
flag = ocif_('arg1')
```

or

```
if( ocif_('arg1') ) goto 100
```

Only one CIF may be open at a time. If another CIF needs to be read, and the old data items are not required, the purge_ command is used to close down the existing *CIFtbx* data tables and the CIF. For example,

```
call purge_
...
if( ocif_('next.cif') ) goto 200
```

If the old CIF data items need to be retained and used in conjunction with items from a new CIF, the following coding would be typical.

```
append_ = .true.
...
if( ocif_('next.cif') ) goto 200
```

5.3. data_

```
logical function data_ (name)
     character name*(*)
```

5.3.1 Definition

data_ is a *logical function* for selecting the block of data items to be read. The function is returned as .true. if the requested data block is found. Only one block may be accessed at a time, and each data_ command removes information of the prior data block. The data_ command initiates any data checking which may have been requested with a prior dict_ command, and data blocks that are not requested will not be checked.

The command argument is:

No.	*Argument*	*Description*
1	<data block name>	data block name string *in quotes*. If this is blank, accept the next encountered data block and place the name in the variable `bloc_`.

5.3.2 Application

The `data_` command can be applied in several different ways. Typically the coding of this function is

```
            if( data_('arg1') ) goto 100
            write(6,'(a)') ' Requested data block not found.'
            stop
      or
            'if( .not.data_(' ') ) goto 1000
            write(6,'(a, a)') ' Data block read: ',bloc_
```

The last statement is typical if the requests for data blocks are in a loop and need to identified.

Here is a simple sequence for opening the file `test.cif` and selecting the data block to be processed.

```
         character*32  name
C
C....... Open the CIF to be accessed
C
100      name='test.cif'
         write(6,'(/2a/)')  ' Read data from CIF  ',name
         if(ocif_(name))      goto 120
         write(6,'(a///)') ' >>>>>>>>> CIF cannot be opened'
         stop
C
C....... Assign the data block to be accessed
C
120      if(data_(' '))       goto 130
         write(6,'(/a/)') ' >>>>>>> No data_ statement found'
         stop
130      write(6,'(/a,a/)') ' Access items in data block  ',bloc_
```

5.4. bkmrk_

```
logical function bkmrk_ (mark)
     integer mark
```

5.4.1 Definition

`bkmrk_` is a *logical function* for saving, or restoring, the current position of a data item in the data block. It permits data elsewhere in the block to be accessed without losing the current position. This function is returned as `.true.` in the save mode provided the internal array `ibkmrk` can hold the current position, and

`.true.` in the restore mode if the bookmark number specified was valid. If `ibkmrk` overflows, increase the parameter `MAXBOOK` in `ciftbx.sys`.

The command argument is:

No.	*Argument*	*Description*
1	<integer variable>	If specified as 0, the variable is returned with a bookmark number pointing to the current location in the data block. If specified as non-zero, the data pointer is set to that position and the integer variable returned as zero.

`bkmrk_` saves the current position in the data block if *arg1* is zero, and then returns the bookmark number in the argument variable. If the integer variable argument is non-zero, the command will restore the position saved for the bookmark number given. The bookmark and the argument are cleared. The position set on return allow reprocessing of the data item or loop row last processed when the bookmark was placed. All bookmarks are cleared by a call to `data_`.

5.4.2 Application

`bkmrk_` can be applied in a variety of ways. Typically this function is used in the save mode as:

```
            integer iset, imark
            ...
            iset = 0
            if( bkmrk_(iset) ) goto 100
            write(6,'(a)') ' Bookmark array has been exceeded.'
            stop
      100   imark = iset
```

and in restore mode as:

```
            iset = imark
            if( bkmrk_(iset) ) goto 200
            write(6,'(a)') ' Bookmark entry does not exist.'
            stop
```

5.5. `find_`

```
logical function find_ (name, type, strg)
     character name*(*), type*(*), strg*(*)
```

5.5.1 Definition

`find_` is a *logical function* used to locate a requested item, name or value, in the current data block, and to return the appropriate string as an argument. The function is returned as `.true.` if the item is found.

The command arguments are:

No.	*Argument*	*Description*	
1	<data name>	The item to be positioned at in quotes. A blank implies the next item encountered.	
2	<string type code>	'head'	reposition at start of CIF
		'name'	reposition at the item name
		'valu'	reposition at the item value
		' '	reposition at next string
3	<returned string >	returned character string specified.	

5.5.2 Application

`find_` may be applied in a variety of ways. Typically the application is as

```
character string*(MAXBUF)
...
if( find_('_atomic_number', 'valu', string) ) goto 100
write(6,'(a)') ' atomic number not found')
```

5.6. `test_`

```
logical function test_ (name)
     character name*(*)
```

5.6.1 Definition

`test_` is a *logical function* used to identify the data attributes of a data item in the current data block. The function is returned as `.true.` if the item is found. The data attributes are stored in the common variables `list_`, `type_`, `dictype_`, `diccat_` and `dicname_`. The values in `dictype_`, `diccat_` and `dicname_` are valid even if the data item is not found in the data block, provided it is present in the dictionaries loaded with the `dict_` command. The data name in the input CIF (as opposed to any of the possible aliased names) is stored in `tagname_`. The line positions of the name and values are stored in `posnam_` and `posval_` and for numbers, the position of the decimal point is stored in `posdec_`. The quotation mark, if any, used is stored in `quote_`.

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	The item to be tested at in quotes. A blank implies the next item encountered.

The *CIFtbx* variables returned are as follows

`list_` is an integer variable containing the sequential number of the loop block in the data block. If the item is not within a loop structure this value will be zero.

type_ is a character*4 variable with the possible values:

'numb'	for number data
'char'	for character data
'text'	for ext data
'null'	if data missing or '?' or '.'

dictype_ is a character*(NUMCHAR) variable with the type code given in the dictionary entry for the named data item. If no dictionary was used, or no type code was specified, this field will simply agree with type_. If a dictionary was used, this type may be more specific than the one given by type_.

diccat_ is a character*(NUMCHAR) variable with the category of the named data item, or '(none)'

dicname_ is a character*(NUMCHAR) variable with the name of the data item that is found in the dictionary for the named data item. If alias_ is .true., this name may differ from the name given in the call to test_. If alias_ is .false. or no preferred alias is found, dicname_ agrees with the data item name.

tagname_ is a character*(NUMCHAR) variable with the name of the data item as found in the input CIF. It will be blank if the data item name requested is not found in the input CIF and may differ from the data item name provided by the user if the name used in the input CIF is an alias of the data item name and alias_ is .true.

posnam_, posval_ and **posdec_** are integer variables that may be examined if information about the horizontal position of the name and data read are needed. posnam_ is the starting column of the data name found (most often 1). posval_ is the starting column of the data value. If the field is numeric, then posdec_ will contain the effective column number of the decimal point. For whole numbers, the effective position of the decimal point is one column to the right of the field.

quote_ is a character*1 variable that may be examined to determine if a quotation character was used on character data.

5.6.2 Application

test_ may be applied in a variety of ways. Here is one coding of this function.

```
            if( .NOT.test_('arg1') ) goto 300
            if( type_ .NE. 'char' )  goto 100
            flag = char_(tagname_, string)
            goto 200
      100   if( type_ .NE. 'numb' )  goto 120
            flag = numb_(tagname_, value, esdval)
            goto 200
```

The following coding is from supplied test example application tbx_ex.f. This shows how a list of data requests in the file 'test.req' may be handled. The

function test_ is used to type the requested item, and then numb_ and char_ are used to get the data values.

```
C
C....... Loop over the data request file
C
         open(unit=8,file='test.req',status='old')
300      read(8,'(a)',end=400) name
C
         f1 = test_(name)
         write(6,'(/a,3x,a,2i5)') name,type_,long_,list_
C
         if(type_.ne.'numb')      goto 320
         f1 = numb_(name, numb, sdev)
         write(6,'(2f10.4)') numb,sdev
         goto 300
C
320      if(type_.ne.'char')      goto 340
         f1 = char_(name, line)
         write(6,'(a)') line(1:long_)
         goto 300
C
340      if(type_.ne.'text')      goto 300
350      f1 = char_(name, line)
         write(6,'(a)') line
         if(text_)  goto 350
         goto 300
```

5.7. name_

```
logical function name_ (name)
     character name*(*)
```

5.7.1 Definition

name_ is a *logical function* used to identify the next data name in the current data block, and to return that string as an argument. The function is returned as .true. if another item is found in the current data block.

The command arguments are:

No.	*Argument*	*Description*
1	<RETURNED data name>	The next encountered data name.

5.7.2 Application

name_ may be applied in a variety of ways. Typically the application is

```
character string*(MAXBUF)
...
flag = name_(string)
flag = test_(string)
```

5.8. numb_

```
logical function numb_ (name, numb, sdev)
     character name*(*)
     real numb, sdev
```

5.8.1 Definition

numb_ is a *logical function* used to read the numerical value and esd value (if appended) of a specified data name in the current data block. The function is returned as .true. if the item is found and is a number. If .false. arguments 2 and 3 are unchanged. If the esd is not attached to the number argument 3 is unaltered.

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	Specified data name of item.
2	<real number variable>	Returned number value of item.
3	<real number variable>	Returned esd of the number value.

5.8.2. Application

numb_ is usually applied as follows.

```
logical   flag
real      value, esdval
character name*(MAXBUF)
...
flag = numb_(name, value, esdval)
```

5.9. numd_

```
logical function numd_ (name, numb, sdev)
     character name*(*)
     double precision numb, sdev
```

5.9.1. Definition

numd_ is a *logical function* used to read a double-precision number and esd value (if appended) of a specified data name in the current data block. The function is returned as .true. if the item is found and is a number. If .false. arguments 2 and 3 are unchanged. If the esd is not attached to the number argument 3 is unaltered.

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	The specified data name of item.

2	<dp number variable>	Returned number value of item.
3	<dp number variable>	Returned esd of the number value of item.

5.9.2 Application

numd_ is typically used as follows.

```
logical   flag
double precision  dvalue, desdval
character name*(MAXBUF)
...
flag = numd_(name, dvalue, desdval)
```

5.10. cmnt_

```
logical function cmnt_ (strg)
     character strg*(*)
```

5.10.1 Definition

cmnt_ is a *logical function* used to read the next comment string in the current data block. The function is returned as .true. if a comment is found. The initial comment character "#" is not included in the returned string. A completely blank line is treated as a comment.

The command arguments are:

No.	*Argument*	*Description*
1	<Returned char variable>	The returned character string of length long_.

5.10.2 Application

cmnt_ is usually applied as follows.

```
logical   flag
character coment*(MAXBUF)
...
flag = cmnt_(coment)
```

5.11. purge_

```
subroutine purge_
```

5.11.1 Definition

purge_ is a subroutine used to reset all data tables and close opened *CIFtbx* files.

5.11.2 Application

`purge_` is called after all data access and output activities are complete.

```
call purge_
```

CHAPTER 6

Reference: Write Functions

6.1. Introduction

The functions needed to write a CIF are similar to those described in Chapter 5 to read a CIF. A summary of the *CIFtbx* writing tools is given in Chapter 2.

The basic steps involved in writing CIF data are as follows.

- load the CIFtbx common variables

```
include 'ciftbx.cmn'
logical flag
```

- open the output CIF

```
flag = pfile_(<file name>)
```

The *file name* is entered as blank if items are to be written to an output CIF that has already been opened. If a non-blank *file name* is entered and the named file is already open the value of flag will be returned as .false.

- write the data_ block header line

```
flag = pdata_(<block name>)
```

If a data block of the same name already exists in the output CIF the value of `flag` will be returned as .false.

- write the data items to the CIF

Values are written with `pchar_`, `pnumb_`, `pnumd_` or `ptext_`.

- close the output CIF

```
call close_
```

Must be done after the last data value or comment is written.

6.2. pfile_

```
logical function pfile_ (fname)
     character fname*(*)
```

6.2.1. Definition

`pfile_` is a *logical function* for opening an output file with the specified file name.

The command arguments are:

No.	*Argument*	*Description*
1	<file name>	Blank for use of currently open file

If the filename is entered as blank, items will be written to an output CIF that has already been opened. If a non-blank file name is entered, and the named file is already open, the value of flag will be returned as .false.

6.2.2. Application

`pfile_` is typically applied as follow.

```
character*(MAXBUF)   fname
logical              flag
...
flag = pfile_(fname)
```

6.3. pdata_

```
logical function pdata_ (name)
     character name*(*)
```

6.3.1. Definition

`pdata_` is a *logical function* for writing a data block header line into the output CIF. The value of the function is returned as .true. if the block is created. The value will be .false. if the block name already exists in the output CIF. This function produces a save frame instead of a data block if the variable `saveo_` is true during the call. No block duplicate check is made for a save frame.

The command arguments are:

No.	*Argument*	*Description*
1	<block name>	Blank for use of the next data block

6.3.2. Application

`pdata_` is usually applied as follows.

```
character*(MAXBUF)   bname
logical              flag
...
flag = pdata_(bname)
```

The following example code is used to open a new output CIF and to write the first `data_` block header.

```
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
C....... Insert a data block code
450      f1 = pdata_('whoops_a_daisy')
```

6.4. ploop_

```
logical function ploop_ (name)
     character name*(*)
```

6.4.1. Definition

ploop_ is a *logical function* for writing a data name belonging to a list of items (i.e. a loop_) being written to the output CIF. The value of the function is returned as .true. if the application conforms with the list syntax.

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	Data name of the item that is a member of a list of items.

6.4.2. Application

ploop_ is usually applied as follows.

```
character*(MAXBUF)   tname
logical        flag
...
flag = ploop_(tname)
```

Here is example code showing how data names, and data values, are added to an output CIF.

```
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=1,10
         f1 = pchar_(' ',alpha(1:i))
         f1 = pnumb_(' ',float(i),float(i)*0.1)
470      f1 = pnumb_(' ',float(i)*8.64523,0.)
```

The above code produces the following output. Note that the loop_ header was created automatically by the first ploop_ call.

```
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
       abcd            4.0(4)          34.5809
       abcde           5.0(5)          43.2262
       abcdef          6.0(6)          51.8714
       abcdefg         7.0(7)          60.5166
       abcdefgh        8.0(8)          69.1618
       abcdefghi       9.0(9)          77.8071
       abcdefghij      10.0(10)        86.4523
```

6.5. pchar_

```
logical function pchar_ (name, string)
     character name*(*), string*(*)
```

6.5.1. Definition

pchar_ is a *logical function* for writing a character string to the output CIF. The value of the function is returned as .true. if the name is unique AND if dict_ is invoked, the name is defined in the dictionary AND the invocation conforms to the CIF logical structure. The action of pchar_ is modified by the variables pquote_ and nblanko_. If pquote_ is non-blank, it is used as a quotation character for the string written by pchar_. The valid values are '"', '"', and ';'. In the last case a text field is written. If the string contains a matching character to the value of quote_, or if quote_ is not one of the valid quotation characters, a valid, non-conflicting quotation character is used. Except when writing a text field, if nblanko_ is true, pchar_ converts a blank string to an unquoted period.

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	If the name is blank, do not output name.
2	<character string>	A character string of MAXBUF chars or less.

6.5.2. Application

pchar_ is typically applied as follows.

```
character*(MAXBUF)   tname
character*(MAXBUF)   string
logical              flag
...
flag = pchar_(tname,string)
```

6.6. pcmnt_

```
logical function pcmnt_ (string)
     character string*(*)
```

6.6.1. Definition

pcmnt_ is a *logical function* for writing a comment; string to the output CIF. The value of the function is returned .true. The comment character "#" *should not* be included in the string. A blank comment is presented as a blank line without the leading "#".

The command arguments are:

No.	*Argument*	*Description*
1	<character string>	A character string of MAXBUF chars or less.

6.6.2. Application

pcmnt_ is typically applied as follows.

```
character*(MAXBUF)   string
logical        flag
...
flag = pcmnt_(string)
```

Here is and example of how two comments are written to the output file.

```
f1 = pcmnt_(' ')
f1 = pcmnt_(' Loops with various numeric and esd formats')
```

6.7. pnumb_

```
logical function pnumb_ (name, numb, sdev)
     character name*(*)
     real numb, sdev
```

6.7.1. Definition

pnumb_ is a *logical function* for writing a single precision number and its esd; to the output CIF. The value of the function is returned as .true. if the name is unique AND if dict_ is invoked, the name is defined in the dictionary AND the invocation conforms to the CIF logical structure. The number of esd digits is controlled by the variable esdlim_

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	If the name is blank, do not output name.

2 <real variable> Number to be inserted.

3 <real variable> Esd number to be appended in parentheses.

6.7.2. Application

pnumb_ is usually applied as follows.

```
character*(MAXBUF)   tname
real           xnumb, xesd
logical        flag
...
flag = pnumb_(tname, xnumb, xesd)
```

6.8. pnumd_

```
logical function pnumd_ (name, numb, sdev)
     character name*(*)
     double precision numb, sdev
```

6.8.1. Definition

pnumd_ is a *logical function* for writing a double precision; number and its esd; to the output CIF. The value of the function is returned as .true. when the name is unique, AND, if dict_ is invoked, the name is defined in the dictionary, AND the invocation conforms to the CIF logical structure. The number of esd digits is controlled by the variable esdlim_

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	If the name is blank, do not output name.
2	<double precision variable>	Number to be inserted.
3	<double precision variable>	Esd number to be appended in parentheses.

6.7.2. Application

pnumb_ is usually applied as follows.

```
character*(MAXBUF)   tname
double precision     xnumb, xesd
logical        flag
...
flag = pnumd_(tname, xnumb, xesd)
```

6.9. ptext_

```
logical function ptext_ (name, string)
     character name*(*), string*(*)
```

6.9.1. Definition

pnumd_ is a *logical function* for writing a character stringcharacter data; to the output CIF. The logical function returned as .true. if the name is unique AND if dict_ is invoked, the name is defined in the dictionary AND the invocation conforms to the CIF logical structure. ptext_ is invoked repeatedly until the text is finished. Only the first invocation will insert a data name.

The command arguments are:

No.	*Argument*	*Description*
1	<data name>	If the name is blank, do not output name.
2	<character string>	A character string of MAXBUF chars or less.

6.9.2. Application

ptext_ is usually applied as follows.

```
character*(MAXBUF)   tname
character*78   string(10)
logical        flag
integer        ii
...
do ii = 1,10
flag = ptext_(tname, string(ii))
enddo
```

Here is code to write three lines of text.

```
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.')
```

This produces the following CIF output. Note the comment created by *CIFtbx* because the tag is not in the dictionary being used.

```
_audit_creation_record                  #< not in dictionary
;Text data may be
 entered like this
 or in a loop.
```

6.10. prefx_

```
logical function prefx_ (strg, lstrg)
```

```
character strg*(*)
integer lstrg
```

6.10.1. Definition

`prefx_` is a *logical function* for writing a prefix; string to subsequent lines of the output CIF. The logical function returned as .true. The second argument may be zero to suppress a previously used prefix, or greater than the non-blank length of the string to force a left margin. Any change in the length of the prefix string flushes pending partial output lines, but does *not* force completion of pending text blocks or loops. This function allows the CIF output functions to be used within what appear to be text fields to support annotation of a CIF.

The command arguments are:

No.	*Argument*	*Description*
1	<character string>	A character string of MAXBUF chars or less.
2	<integer variable>	The length of the prefix string to use.

6.10.2. Application

`prefx_` is usually applied as follows.

```
character*10   string
integer        lstr
logical        flag
...
lstr = len(string)
flag = prefx_(string, lstr)
```

6.11. `close_`

`subroutine close_`

6.11.1. Definition

`close_` is a subroutine called to close the creation CIF.

6.11.2. Application

`close_` must be used if `pfile_` is used.

```
call close_
```

CHAPTER 7

Reference: Variables

Most simple *CIFtbx* applications need be aware of just a few basic variables: `list_`, `loop_`, `strg_`, `text_`, and `type_`. For advanced applications, a rich environment of general monitor, general control, input monitor and output control variables is provided.

General Monitor	General Control	Input Monitor	Output Control
	alias_		aliaso_
			align_
	append_		
		bloc_	
		decp_	pdecp_
		dictype_	
		diccat_	
		dicname_	
		dicver_	
		esddig_	pesddig_
			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_	

In order to make any of these variables available to a program, function or subroutine, the common block `ciftbx.cmn` must be included, as by

```
include 'ciftbx.cmn'
```

The following *CIFtbx* general monitor variables give the status of general processing by *CIFtbx*.

`file_` *Character*(MAXBUF) variable*: Set by `dict_`, `ocif_` and `pfile_` to the filename of the current file.

`longf_` *Integer variable*: Set by `dict_`, `ocif_` and `pfile_ to` the length of the filename in file_.

`precn_` *Integer variable*: Reports the record number (starting from 1) of the last line written to the output CIF. Set to zero by `init_`. Also set to zero by `pfile_` and `close_` if the output CIF file name was not blank.

`recn_` *Integer variable*: Reports the record number (starting from 1) of the last line read from the direct access copy of the input CIF.

`tbxver_` *Character*32 variable:* Initialized to the *CIFtbx* version and date in the form `'CIFtbx version N.N.N, DD MMM YY '`[6]

The following *CIFtbx* general control variables control input and general processing by *CIFtbx*. The user may accept the default values or may store new values into these variables to change the behavior of the commands.

`alias_` *Logical variable*: Set `.true.` will cause calls to *CIFtbx* functions to accept aliases of data item names (see 2.6, above). The preferred synonym from the dictionary will be substituted internally, provided aliased data names were supplied by an input dictionary (via `dict_`). The default is `.true.`, but `alias_` may be set to `.false.` in an application.

`append_` *Logical variable*: Set `.true.` will cause each call to `ocif_` to append the information found to the current CIF to the infoormation in the direct access file for any prior CIFs. The default is `.false.`, in which case each call to `ocif_` overwrites the information for the prior CIF.

`line_` *Integer variable:* Specifies the input/output line limit for processing a CIF. The default value is 80 characters. This limit includes the visible printable characters of the line, not the system-dependent line terminators. This variable may be set by the program. The maximum value is `MAXBUF` that has a default value of 200.

---

[6]The length of this string may be increased in future versions to allow for multidigit version numbers and to become year-2000 compliant,

nblank_ *Logical variable*: Set `.true.` will cause calls to `char_` or `test_` that encounter a non-text quoted blank to return the type as '`null`' rather than '`char`'.

recbeg_ *Integer variable*: Gives 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*: Gives 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:* Set `.true.` causes tab character expansion to blanks during the reading of a CIF. The default is `.true.`

The following *CIFtbx* input monitor variables monitor the processing of input from a CIF.

bloc_ *Character*(NUMCHAR) variable*: Set by `data_ to` the current block name.

decp_ *Logical variable:* Set when processing numeric input, `.true.` if there is a decimal point in the numeric value, `.false.` otherwise, set by `numb_` and `numd_`.

diccat_ *Character*(NUMCHAR) variable*: Set by `test_`, `find_`, `char_`, `numb_`, `numd_` to the category (see `test_`) or '`(none)`'

dicname_
Character(NUMCHAR) variable*: Set by `test_`, `find_`, `char_`, `numb_`, `numd_` to the root alias (see `test_`) of name or by `dict_` to the name the dictionary just loaded.

dictype_
Character(NUMCHAR) variable*: Set to the precise data type code (see `test_`). This is the type code given in the dictionary entry for the named data item. If no dictionary was used, or no type code was specified, this field will simply agree with `type_`. If a dictionary was used, this type may be more specific (e.g. `'float'` or `'int'`) than the one given by `type_` (e.g. '`numb`').

dicver_ *Character*(NUMCHAR) variable*: Set by `dict_` to the version of the dictionary

esddig_ *Integer variable* : Set when processing numeric input to the number of esd digits in the last number read from a CIF. Will be zero if no esd was given.

glob_ *Logical variable*: Set by `data_` to signal that the current data block is actually a global block (`.true.` for a global block). The application is responsible for managing the relationship of global data to other data blocks.

list_ *Integer variable*: Set by test_, find_, char_, numb_, numd_ to the sequential number of the loop block in the data block. If the item is not within a loop structure this value will be zero.

long_ *Integer variable*: Set by test_, find_, char_, cmnt_, numb_, numd_ to the length of the data string in strg_.

loop_ *Logical variable*: Set by test_, find_, char_, numb_, numd_ to .true. if another loop packet is present.

lzero_ *Logical variable*: Set when processing numeric input to .true. if the numeric value is of the form [sign]0.nnnn rather than [sign].nnnn, .false. otherwise, set by numb_ and numd_.

posdec_ *Integer variable* : Set when processing numeric input to the column number (position along the line, counting from 1 at the left) of the decimal point or the last number read, if a decimal point was present, set by numb_ and numd_. For whole numbers, the effective position of the decimal point is one column to the right of the field.

posend_ *Integer variable* : Set by test_, find_, char_, numb_, numd_ to the ending column number (position along the line, counting from 1 at the left) of the last data value read, not including a terminal quote.

posnam_ *Integer variable*: Set by test_, find_, char_, numb_, numd_ to the starting column number (position along the line, counting from 1 at the left) of the last name or comment or data block read.

posval_ *Integer variable*: Set by test_, find_, char_, numb_, numd_ to the starting column of the last data value read. Also reports the column number (position along the line, counting from 1 at the left) of the terminal "save_" of a save frame.

quote_ *Character variable*: Set by test_, find_, char_ to the quotation symbol found delimiting the last string read.

save_ *Logical variable* : Set by data_ to .true. if the current data block is actually a save-frame.

strg_ *Character*(MAXBUF) variable*: Set by test_, find_, char_, cmnt_, numb_, numd_ to the current data item.

tabx_ *Logical variable*: Set .true. will cause the expansion of tab characters to blanks during the reading of a CIF. The default is .true.

tagname_ *Character*(NUMCHAR) variable*: Set by test_, find_, char_, numb_, numd_ to the name of the data item as found in the input CIF. It will be blank if the data item name requested is not found in the input CIF

and may differ from the data item name provided by the user if the name used in the input CIF is an alias of the data item name and `alias_` is `.true.`

`text_` *Logical variable*: Set by `test_`, `find_`, `char_` to `.true.` if another text line is present.

`type_` *Character*4 variable*: Set by `test_`, `find_`, `char_`, `numb_`, `numd_` to the data type code (see `test_`), i.e. to
'numb' for number data
'char' for character data
'text' for text data
'null' if data missing or '?' or '.'

The following *CIFtbx* output control variables control output formats of a CIF being written.

`aliaso_` *Logical variable*: Set `.true.` will cause output functions to convert alias names to preferred synonyms in the dictionary. The default is `.false.` The setting of `aliaso_` is independent of the setting of `alias_`.

`align_` *Logical variable*: Set `.true.` will cause an alignment of `loop_` lists during the creation of a CIF. The default is `.true.`

`esdlim_` *Integer variable*: Specifies the upper limit of esd's produced by `pnumb_`, and, implicitly, the lower limit. The default value is 19, which limits esd's to the range 2-19. Typical values of `esdlim_` might be 9 (limiting esd's to the range 1-9), 19, or 29 (limiting esd's to the range 3-29)

`globo_` *Logical variable*: Set `.true.` will cause the output data block from `pdata_` to be written as a global block.

`nblanko_` *Logical variable*: Set `.true.` will cause the output functions to convert quoted *blank* strings to an unquoted period (i.e. to a data item of type *null*). The default is `.false.`

`pdecp_` *Logical variable*: Set `.true.` will cause the output functions to insert a decimal point in all numbers written by `pnumb_` or `pnumbd_`. If set `.false.` then a decimal point will be written only when needed. The default is `.false.`

`pesddig_`
Integer variable: Specifies the the number of digits of esd's produced by `pnumb_` and `pnumd_`, provided the value of `pesddig_` is non-zero and `esdlim_` is negative.

`plzero_` *Logical variable*: Set `.true.` will cause the output functions to insert a zero before a leading decimal point. The default is `.false.`

pposdec_
Integer variable: Specifies the column number (position along the line, counting from 1 at the left) of the decimal point for the next number to be written. This acts very much like a decimal centered tab in a word processor, to help align columns of number on a decimal point, if a decimal point is present.

pposend_
Integer variable: Specifies the ending column number of the next number or quoted character value to be written. Used to pad with zeros or blanks.

pposnam_
Integer variable: Specifies the starting column number of the next name or comment to be written.

pposval_
Integer variable: Specifies the starting column number of the next data value to be written.

pquote_ *Character variable*: Specifies the quotation symbol to be used for the next string written.

precn_ *Integer variable*: Returns the record number of the last line written to the output CIF. Set to zero by `init_`. Also set to zero by `pfile_` and `close_` if the output CIF file name was not blank.

ptabx_ *Logical variable*: Set `.true.` will cause tab character expansion to blanks during the creation of a CIF. The default is `.true.`

saveo_ *Logical variable*: Set `.true.` will cause the `pdata_` to output a save frame header line rather than a data block header line. The default is `.false.`

tabl_ *Logical variable*: Set `.true.` if tab-stop alignment is to be used for data items written to the CIF. The default is `.true.`

Column Numbering

The columns of both input and output CIFs are numbered from 1 on the left, usually to 80, unless `line_` has been changed to a different value. Variables such as `pposdec_`, `pposend_`, `pposnam_`, and `pposval_`, which specify the column numbers at which items are to be written may be set to zero if *CIFtbx* is to use its best estimate. In general, the routines that accept the column number specifications will reset the associated variables to zero after each use. The following CIF fragment illustrates the column numbering used.

```
# This is a CIF fragment to illustrate CIFtbx column numbering
#        1         2         3         4         5         6         7
#2345678901234567890123456789012345678901234567890123456789012345678901234567890

    _cell.length_a     37.58
#   ^                  ^ ^ ^
#   |                  | | |
#   posnam_ = 5        | | posend_=28
#                      | |
#                      | posdec_=26
#                      |
#                      posval_= 24
```

CHAPTER 8

Reference: Error Message Glossary

The *CIFtbx* functions avoid issuing error messages unless they are requested, or there is no sensible way to continue. The preferred approach is for the *CIFtbx* functions to return *true* or *false* function values so that the application program can respond to run-time problems in a controlled way and take corrective action if it is possible. Nevertheless some types of processing errors, such as exceeding the dimensions of critical *CIFtbx* arrays, will require that an appropriate message be issued and execution cease.

All *CIFtbx* error messages have a common format. Each begins with either a "warning" or "error" header line with the name of the file being processed, the current data block or save frame, and the line number. The next line contains the text of the message.

For example, the following warning message is issued by the test program `tbx_ex.f` when it finds that the file `test.cif` contains the unknown data item name `_dummy_test`, in the data block `data_mumbo_jumbo`.

```
ciftbx warning:  test.cif    data_mumbo_jumbo   line: 12
Data name _dummy_test not in dictionary!
```

Fatal Errors: Array Bounds

The following 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 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

The following array bounds message is not fatal.

More than MAXBOOK bookmarks requested
The number of simultaneous bookmarks is more than MAXBOOK. The function bkmrk_ will return .false., and processing will continue, so that the calling program may take appropriate action before termination, but this is effectively a fatal error for which recompilation with a larger value of MAXBOOK is necessary.

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 a "." 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).

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

Fatal Errors: Invalid Arguments

The following messages are generated by calls with invalid arguments.

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

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 it more likely indicates a typographical error in the dictionary or the CIF.

`Category` *`<key-tag>`* **`not given`**
The category key tag specified in a DDL2 data set or the list reference tag specified in a DDL1 data set was not found, even though some tags in that category were given. This is usually due to an incomplete loop header.

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_, save_ or global_ was found when expected.

Duplicate data item <*name*>
There were two or more identical data names <*name*> 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 out of the range that can be processed on this machine. If the string involved is not intended to be processed as a number, then quoting it may resolve the problem.

Heterogeneous categories in loop <*new cat-code*> **vs** <*old cat-code*>
Looped lists should not contain data from different categories. This message is issued if the category of a new data items 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. No warning is issued on this level for a loop for which all data items have no category declared.

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, which is the upper bound for lines in a valid CIF. 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
When writing an output CIF, a looped list of 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.

Warnings: Output Errors

Converted pchar_ output to text for <*string*>
An attempt was made to write a string to a CIF with pchar_ instead of ptext_, but the string contains a combination of characters for which ptext_ must be used. This warning is issued and processing continues.

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 esd that cannot be presented properly on this machine. A bounding value (0 or 99999 is used for the esd, this warning message is issued and processing continues.

Invalid value of esdlim_ reset to 19
In processing numeric output, a value of esdlim_ less than 9 or greater than 99999 was found. The value of esdlim_ is forced to 19, this warning is issued, and processing continues.

Missing: missing loop_ name set as _DUMMY
Missing: missing loop_ items set as DUMMY
In processing a loop_ the library has had to pad either the header or the values. This warning is issued and processing continues.

Output CIF line longer than line_
In processing a line of output the data being presented forces the line to be longer than the value specified in line_. This warning is issued and processing continues. This warning should not occur if tags and values each of which can fit on a line are used.

Out-of-sequence call to end text block
The processing logic for termination of a text block has been invoked without a text block having been started. This warning is issued and processing continues. This can only be caused by users calling some of the internal routines of the library rather than the standard interface routines.

Output prefix may force line overflow
A prefix string specified to prefx_ is longer than the maximum line length allowed less the allowed length of tags. This warning is issued and processing continues.

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

Warnings: Dictionary Checks

Aliases and names in different loops; only using first alias
When a DDL2 dictionary contains a loop of alias declarations the corresponding data name declarations are expected to be in the same loop. This message is issued if separate loops are used. Only the first alias name is used, but processing continues.

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, these warnings are issued. The redeclaration is ignored.

Categories and names in different loops
Types and names in different loops
When a DDL2 dictionary contains a loop of category or type declarations, the corresponding data name declarations are expected to be in the same loop. This message is issued if separate loops are used. Only the first category name or type is used, but processing continues.

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
When a DDL2 dictionary contains a new declaration of an alias for a data name that is in conflict with a previous alias definition, this warning is issued. The second alias declaration is ignored.

Duplicate definition of same alias
When a DDL2 dictionary contains a new declaration of an alias for a data name that duplicates a previously defined alias pair, this warning is issued.

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 initial characters of the item name, this warning is issued and processing continues. This may indicate a typo or a deprecated item in the dictionary.

Item type <*type-code*> **not recognised**
In *CIFtbx*, DDL2 dictionary precise type codes are translated to the DDL primitive type codes `numb`, `char` and `text`. If an unrecognised type code is found for which *CIFtbx* does not have a translation, this warning is issued.

Multiple DDL 1 and 2 category definitions
Multiple DDL 1 and 2 related key definitions
Multiple DDL 1 and 2 name definitions
Multiple DDL 1 and 2 type definitions
Multiple DDL 1 and 2 related item definitions
Multiple DDL 1 and 2 related item functions
These messages are issued if both DDL and DDL2 style declarations for categories, category keys (i.e. list references), 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
These messages are issued if 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 and processing continues.

No category defined in block <*name*> **and name** <*name*> **does not match**
This message is issued if 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 message usually indicates a typographical error in the dictionary.

No category specified for name <*name*>
This warning is issued if 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*>
This message is issued if 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 of 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.

APPENDIX A

Usage Restrictions and Policy

Creative endeavors depend on the lively exchange of ideas. There are laws and customs that establish rights and responsibilities for authors and the users of what authors create. This notice is not intended to prevent you from using the software and documents in this package, but to ensure that there are no misunderstandings about terms and conditions of such use.

Please read the following notice carefully. If you do not understand any portion of this notice, please seek appropriate professional legal advice before making use of the software and documents included in this software package. In addition to whatever other steps you may be obliged to take to respect the intellectual property rights of the various parties involved, if you do make use of the software and documents in this package, please give credit where credit is due by citing this package, its authors and the URL or other source from which you obtained it, or equivalent primary references in the literature with the same authors.

Some of the software and documents included within this software package are the intellectual property of various parties, and placement in this package does not in any way imply that any such rights have in any way been waived or diminished.

With respect to any software or documents for which a copyright exists, **ALL RIGHTS ARE RESERVED TO THE OWNERS OF SUCH COPYRIGHT**.

Even though the authors of the various documents and software found here have made a good faith effort to ensure that the documents are correct and that the software performs according to its documentation, and we would greatly appreciate hearing of any problems you may encounter, the programs and documents any files created by the programs are provided **AS IS** without any warrantee as to correctness, merchantability or fitness for any particular or general use.

THE RESPONSIBILITY FOR ANY ADVERSE CONSEQUENCES FROM THE USE OF PROGRAMS OR DOCUMENTS OR ANY FILE OR FILES CREATED BY USE OF THE PROGRAMS OR DOCUMENTS LIES SOLELY WITH THE USERS OF THE PROGRAMS OR DOCUMENTS OR FILE OR FILES AND NOT WITH AUTHORS OF THE PROGRAMS OR DOCUMENTS.

The IUCr Policy
on the
Use of the Crystallographic Information File (CIF)

(Reproduced with permission from the IUCr Web Page)

The Crystallographic Information File [Hall, Allen, Brown 91] is, as of January 1992, the recommended method for submitting publications to Acta Crystallographica Section C. The International Union of Crystallography holds the Copyright on the CIF, and has applied for Patents on the STAR File syntax which is the basis for the CIF format.

It is a principal objective of the IUCr to promote the use of CIF for the exchange and storage of scientific data. The IUCr's sponsorship of the CIF development was motivated by its responsibility to its scientific journals, which set the standards in crystallographic publishing. The IUCr intends that CIFs will be used increasingly for electronic submission of manuscripts to these journals in future. The IUCr recognises that, if the CIF and the STAR File are to be adopted as a means for universal data exchange, the syntax of these files must be strictly and uniformly adhered to. Even small deviations from the syntax would ultimately cause the demise of the universal file concept. Through its Copyrights and Patents the IUCr has taken the steps needed to ensure strict conformance with this syntax.

The IUCr policy on the use of the CIF and STAR File processes is as follows:

• 1 CIFs and STAR Files may be generated, stored or transmitted, without permission or charge, provided their purpose is not specifically for profit or commercial gain, and provided that the published syntax is strictly adhered to.

• 2 Computer software may be developed for use with CIFs or STAR files, without permission or charge, provided it is distributed in the public domain. This condition also applies to software for which a charge is made, provided that its primary function is for use with files that satisfy condition 1 and that it is distributed as a minor component of a larger package of software.

• 3 Permission will be granted for the use of CIFs and STAR Files for specific commercial purposes (such as databases or network exchange processes), and for the distribution of commercial CIF/STAR software, on written application to the IUCr Executive Secretary, 2 Abbey Square, Chester CH1 2HU, England. The nature, terms and duration of the licences granted will be determined by the IUCr Executive and Finance Committees.

In summary, the IUCr wishes to promote the use of the STAR File concepts as a standard universal data file. It will insist on strict compliance with the published syntax for all applications. To assist with this compliance, the IUCr provides public domain software for checking the logical integrity of a CIF, and for validating the data name definitions contained within a CIF. Detailed information on this software, and the associated dictionaries, may be obtained from the IUCr Office at 5 Abbey Square, Chester CH1 2HU, England.

APPENDIX B

Installation of *CIFtbx*

Quick Installation

Here is the recommended procedure for implementing *CIFtbx2* summarized for experienced users:

1. Create and populate a directory named `ciftbx_2.6`
   - 1.1. Obtain the ciftbx release kit, `ciftbx.cshar` and place it in `ciftbx_2.6`
   - 1.2. Unpack `ciftbx.cshar`
   - 1.3. Obtain the dictionaries and place them in `ciftbx_2.6/dictionaries` in compressed format
   - 1.4. Make listings
2. Build *CIFtbx2*
   - 2.1. Go to ciftbx.src by '**`cd`** `ciftbx.src`'.
   - 2.2. Adjust Parameters, if necessary
   - 2.3. Make the routines by '**`make`** `all`'.
3. Test *CIFtbx2*
   - 3.1. Run the tests by '**`make`** `tests`'.
   - 3.2. Examine the test output

Detailed Installation Instructions

The terminology we will use in these instructions is for Unix systems, using the C-shell. For non-Unix systems, different file names and system commands will have to be used.

1. Create and populate a directory named `ciftbx_2.6`

Normally, *CIFtbx2* is used partially as a source-code library, for common header code to include, and partially as an object code library for linking. If you are the system manager installing *CIFtbx2* for a group of users, *CIFtbx2* should be installed in a directory to which all users have read access. If you are installing *CIFtbx2* for yourself, create a directory within your own directory tree. During installation it is best to create the directory with a unique name, not conflicting with that of any prior version. We will need to reference the directory you chose, so we will treat it as having been assign to a variable named '`CIFtbx2`', which we will refer to as a Unix-style environment variable '`$CIFtbx2`'. So, if you are system manager, you might

```
cd /usr/local
mkdir ciftbx_2.6
```

```
setenv CIFtbx2 /usr/local/ciftbx_2.6
```

If you are installing *CIFtbx2* for yourself, you might

```
cd
mkdir ciftbx_2.6
setenv CIFtbx2 ~/ciftbx_2.6
```

1.1. Obtain the ciftbx release kit, `ciftbx.cshar` and place it in `ciftbx_2.6`

Release kits for *CIFtbx2* can be obtained from several places. They are available from the IUCr and its mirror sites, the NDB and its mirror sites and from the authors. For each indicated geographical area, the best locations on the Web from which to obtain *CIFtbx2* are:

Europe:

`http://www.iucr.org/iucr-top/cif`
(IUCr, Chester, England)

`http://www.ebi.ac.uk/NDB/mmcif/software`
(NDB mirror site at EBI near Cambridge, England)

`http://www.se.iucr.org/iucr-top/cif`
(IUCr mirror site at University of Stockholm, Stockholm, Sweden)

`http://www.fr.iucr.org/iucr-top/cif`
(IUCr mirror site at University Pierre et Marie Curie, Paris, France)

United States:

`http://ndbserver.rutgers.edu/NDB/mmcif/software`
(NDB, Rutgers University, New Brunswick, New Jersey, USA)

`http://iucr.sdsc.edu/iucr-top/cif`
(IUCr mirror site at San Diego Supercomputing Center, San Diego, California, USA)

`http://www.bernstein-plus-sons.com/software/ciftbx`
(Bernstein + Sons remote web site at Pair Networks, Pittsburgh, Pennsylvania, USA)

South Africa:

`http://www.za.iucr.org;/iucr-top/cif`
(IUCr mirror site at Dept. Structural Chemistry, Univ. Witwatersrand, South Africa)

Asia:

`http://ndbserver.nibh.go.jp/NDB/mmcif/software`
(NDB mirror site at Structural Biology Centre, NIBH Japan)

Australia:

`http://www.crystal.uwa.edu.au/~yaya/software/cc_star.html`
(Crystallography Centre, University of Western Australia, Nedlands, Australia)

The release kit you fetch may be a compressed C-shell archive, `ciftbx.cshar.Z` or a compressed shell archive, `ciftbx.shar.Z`. You only need one of them, but

whichever one you fetch, be certain to uncompress it. You may also fetch the uncompressed kits directly, but that will greatly increase the transmission time. In any case, you want to end up with either `ciftbx.cshar` or `ciftbx.shar` in the directory `$CIFtbx2`.

1.2. Unpack `ciftbx.cshar`

WARNING: ***The files in this kit will unpack into a directory named `ciftbx.src`. It is a good idea to save the current contents of ciftbx.src and then to make the directory empty.***

If you are in a Unix system, unpacking the kit is simple. Follow the instructions at the front of the archive, i.e.

```
cd $CIFtbx2
csh ciftbx.cshar
```

or, if you have `ciftbx.shar`

```
cd $CIFtbx2
sh ciftbx.shar
```

This will create two new directories (`ciftbx.src` and `dictionaries`) as subdirectories of `$CIFtbx2` and install the following files:

`$CIFtbx2/mkdecompln` decompression script used by `Makefile`

`$CIFtbx2/rmdecompln` cleanup script used by `Makefile`

`$CIFtbx2/ciftbx.src/README.ciftbx`
instructions on how to build *CIFtbx*

`$CIFtbx2/ciftbx.src/MANIFEST`
a list of files in the kit

`$CIFtbx2/ciftbx.src/Makefile`
a preliminary control file for make

`$CIFtbx2/ciftbx.src/ciftbx.cmf`
CIFtbx function definitions for applications

`$CIFtbx2/ciftbx.src/ciftbx.cmn`
CIFtbx common for inclusion into applications

`$CIFtbx2/ciftbx.src/ciftbx.cmv`
CIFtbx common variables only

`$CIFtbx2/ciftbx.src/ciftbx.f`
CIFtbx Fortran source

`$CIFtbx2/ciftbx.src/ciftbx.sys`
CIFtbx common for inclusion into ciftbx.f

`$CIFtbx2/ciftbx.src/clearfp.f`
dummy routine holding place for clearfp_sun.f

`$CIFtbx2/ciftbx.src/clearfp_sun.f`
SUN routine to clear floating point exceptions

`$CIFtbx2/ciftbx.src/hash_funcs.f`
hash-table control routines used by CIFtbx

`$CIFtbx2/ciftbx.src/mtest.out`
CIF output by the tbx_exm.f run
`$CIFtbx2/ciftbx.src/mtest.prt`
print file output from tbx_exm.f run
`$CIFtbx2/ciftbx.src/tbx_ex.f`
example application used against cif_core.dic
`$CIFtbx2/ciftbx.src/tbx_exm.f`
example application used against cif_mm.dic
`$CIFtbx2/ciftbx.src/test.cif`
example CIF used by tbx_ex.f
`$CIFtbx2/ciftbx.src/test.out`
CIF output from tbx_ex.f run
`$CIFtbx2/ciftbx.src/test.prt`
print file output from tbx_ex.f run
`$CIFtbx2/ciftbx.src/test.req`
example request file used by tbx_ex.f

Check that each file has been installed in the correct location in the directory tree. If there has been a problem (usually because file with the same name was found) move the offending file to a safe place, empty out the directory and unpack again.

IF YOU DON'T HAVE UNIX

If sh or csh are not available, then it is best to start with the "C-Shell Archive", `ciftbx.cshar` and do the steps that follow. If you must use the "Shell Archive", `ciftbx.shar` you should be aware that the lines you want to extract have been prefixed with "X", while most of the lines you want to discard have not. For a "C-Shell Archive" such prefixes are rare and the file is easier to read. Assume you have a "C-Shell Archive".

Use your editor to separate the different parts of the file into individual files in your workspace. Each part starts with a lot of Unixisms, then several blank lines and then two lines that identify the file, and most importantly, contain the text

"**CUT_HERE_CUT_HERE_CUT_HERE**"

You can look at the line before and the line after to see if you are at the head or tail of a file. Use your editor to search for the "CUT_HERE" lines. Each part is carefully labeled and indicates the recommended filename for the separated file. On some machines these filenames may need to be altered to suit the OS or compiler. (e.g. on MS/DOS PC's you may want to change 'hash_funcs.f' to something like 'hashfunc.for'). Even though this particular release has no lines for which an "X" prefix is used within a file, be warned that, in general, you should look for lines that start with "X" and remove the "X". Put each file in the places noted above.

1.3. Obtain the dictionaries and place them in `ciftbx_2.6/dictionaries` in compressed format

WARNING: ***To test CIFtbx, you must have the dictionaries `cif_core.dic.Z` and `cif_mm.dic.Z` in compressed form installed in a directory named `dictionaries`.***

You can obtain the two dictionaries used for testing over the Web. The latest version of `cif_core.dic`, the core dictionary, can be obtained from the IUCr web server

```
http://www.iucr.ac.uk
```

or its mirror sites, and the latest version of `cif_mm.dic`, the mmCIF dictionary, can be obtained from the mmCIF web page at

```
http://ndbserver.rutgers.edu/NDB/mmcif
```

or its mirror sites. The dictionaries used in the test cases provided with *CIFtbx2* are available from the same sites that provide the release kit.

You may obtain the dictionaries in compressed or uncompressed form, but the standard *CIFtbx2* tests work from the compressed form, so be careful to finish installing the dictionaries by leaving them in compressed form.

1.4. Make Listings

Once you have separated out these files, list `ciftbx.f`, `Makefile`, `hash_funcs.f`, `tbx_exm.f` and `tbx_ex.f` in particular (all if possible!) and carefully read the descriptions in the front of these files. Remember that `tbx_ex.f` and `tbx_exm.f` are only examples of CIF applications -- they show how some basic CIF operations can be performed, but they are not necessarily sensible or typical of what an actual application would look like!

WARNING -- if you are running on a SUN, or other system which treats floating point underflows as an error, you may wish to list `clearfp_sun.f`

2. Build *CIFtbx2*

2.1. Go to ciftbx.src by '`cd $CIFtbx2/ciftbx.src`'.

The entire build of *CIFtbx2* is done in the directory `$CIFtbx2/ciftbx.src`, even though some of the files needed are elsewhere. The build instructions assume that the scripts mkdecompln and rmdecompln are in `$CIFtbx2` and that the compressed dictionaries are in `$CIFtbx2/dictionaries`. If you are working on a non-Unix system, vary this according to the requirements of your OS and compiler. You will find it simplest to work if you place the *CIFtbx2* files together in a common subdirectory named `ciftbx.src`. Be very careful if you place them in a directory with other files, since some of the build and test

instructions remove or overwrite existing files, especially with extensions such as `.o`, `.lst`, `.diff` or `.new`.

WARNING: ***If you are running on a SUN or similar system that treats floating point underflow as an error, you may need to use `clearfp_sun.f` Please read the following paragraph carefully.***

Before building the code, you may wish to replace the file `clearfp.f` with code appropriate to your system. The routine is called by *CIFtbx2* to clear possible floating point underflows which may be generated when the code attempts to find the number of digits of precision supported on your system. No special action is required to clear an underflow on many systems, but on a SUN, for example, execution of the code to test machine precision generates messages about underflow and inexact arithmetic. On a SUN, these messages may be avoided by replacing `clearfp.f` by `clearfp_sun.f`. On other machines sensitive to underflow, you may have to use other (usually similar) code.

2.2. Adjust Parameters

For most purposes, the default values of the parameters in *CIFtbx2* are satisfactory. However, when working with very large dictionaries or on computers with limited memory, it may be necessary to adjust some of the parameters that control the sizes of various arrays. If changes are necessary, the parameters can be found in `ciftbx.cmv` and `ciftbx.sys`. In order for changes to take effect, all code should be recompiled after the changes are made.

NUMHASH (default 53)

In the conversion from *CIFtbx* to *CIFtbx2*, there were two issues to address in the changes in size of the dictionary and of names: allocating appropriate storage and preserving efficiency of the code execution. New parameters were introduced for the size-dependent changes, so that future changes can go more smoothly. Efficiency is achieved by extensive use of hash-table-controlled lists. The routines used can be found in `hash_funcs.f`. Ordinarily the user should not have to deal directly with these routines. The only change that might be made for tuning would be to adjust the parameter `NUMHASH` in `ciftbx.sys`. This is presently set 53, which would mean, for up to 3200 names, typical searches for name matches would look at sub-lists to less than approximately 61 names. Greater timing efficiency can be achieved at a slight expense in memory by increasing `NUMHASH` to some larger number. It is recommended that a prime be used for best efficiency in distribution of names among sub-lists.

NUMCHAR (default 48)

The maximum number of characters in a data name is set by NUMCHAR. This is sufficient for all presently available dictionaries, but it is possible that names of up to 76 characters could be defined. If longer names must be processed, increase the parameter.

NUMDICT (default 3200)

The cumulative total number of names in all dictionaries loaded must be less than or equal to NUMDICT. Increase this number if many dictionaries must be loaded simultaneously.

NUMBLOCK (default 500)

CIFtbx2 processes each data block independently and loads information about all the names in the data block currently being processed into arrays, the size of which are controlled by NUMBLOCK. Increase this parameter if a single data block might contain more than 500 names.

NUMLOOP (default 50)

The maximum number of loops in a data block is controlled by the parameter NUMLOOP. Increase this number if any data block to be processed may contain more than 50 loops.

NUMITEM (default 50)

The maximum number of items in a single loop is controlled by the parameter NUMITEM. Increase this number if any loop to be processed may contain more than 50 data item tags in its header.

MAXBUF (default 200)

The maximum number of characters in a line to be read is controlled by the parameter MAXBUF. While valid CIFs can only have 80 columns on a line, *CIFtbx2* can be used to process files with longer lines. Increase MAXBUF if line to be processed may contain more than 200 characters.

NUMPAGE (default 10)
NUMCPP (default 16384)
CIFtbx attempts to reduce the number of disk access to the direct access file by grouping lines into large pages and by holding several pages in memory. The number of memory resident pages of the direct access file is controlled by the parameter NUMPAGE. The number of characters in each page is controlled by the parameter NUMCPP. It is essential that NUMCPP be larger than MAXBUF, i.e. large enough to hold at least one line of the input CIF. It is desirable that NUMCPP be large enough to hold several lines, and that it be a multiple of the block sizes of disk drives used. Naturally, more memory will be required by *CIFtbx* if larger values of NUMPAGE are used, but there will be less demand for disk access.

2.3. Make the routines by '`make all`'.

On a Unix system, most of what you need to do to build and test *CIFtbx2* is laid out in `$CIFtbx2/ciftbx.src/Makefile`. **Be sure to examine and edit this**

file appropriately before using it. But, once properly edited, all you should need to do is

```
make clean
```

to remove old object files, and

```
make all
```

to build new versions of `ciftbx.o`, etc. Later we will

```
make tests
```

to test what you have built. Note that the `Makefile` takes some initial action to force mkdecompln and rmdecompln to be executable. See the section marked `postshar`.

If you don't wish to use the `Makefile` or can't, then here are the essential steps to build *CIFtbx2*:

(a) compile `tbx_ex.f` [note that provided the Fortran "include" function is available to you, the files `ciftbx.f`, `ciftbx.sys`, `hash_funcs.f`, `ciftbx.cmn`, `ciftbx.cmf` and `ciftbx.cmf` will be automatically opened and processed by this single operation]

(b) link `tbx_ex.o` as the executable file `tbx_ex`

(c) compile `tbx_exm.f`, `ciftbx.f`, and `hash_funcs.f`

(d) link `tbx_exm.o`, `ciftbx.o` and `hash_funcs.o` as the executable file `tbx_exm`

3. Test *CIFtbx2*

To execute the supplied example applications `tbx_ex.f` and `tbx_exm.f` identically to the test outputs supplied, a copy of the CIF Core Dictionary `cif_core.dic` and of the macromolecular CIF dictionary version 0.9.01 `cif_mm.dic` must be available in your work area. If they are not the tests will proceed with a warning message but no validations checks will occur. See section 1.3, above. Once you have all both dictionaries, compress them, and edit the definitions of `MMDICPATH` and `COREDICPATH` in `Makefile` to agree with the permanent locations of the formerly uncompressed dictionaries. The `Makefile` will create local soft links to temporary uncompressed copies of the dictionaries. If you can afford the space for permanent uncompressed copies, change the definition to `EXPAND` in `Makefile` to a non-temporary directory, such as '.'

3.1. Run the tests by '`make tests`'.

The tests can be run the your output compared to the expected output by

```
cd $CIFtbx2/ciftbx.src
make tests
```

If you cannot use the `Makefile`, here are the steps to run the tests:

(e) execute `tbx_ex` so that the list file `test.lst` is connected to device 6 (stdout). The input CIF `test.cif` and the output CIF `test.new` will be automatically opened. For a Unix OS the command will look like this:

```
tbx_ex > test.lst
```

(f) execute `tbx_exm` so that the list file `mtest.lst` is connected to device 6 (stdout). The input CIF `test.cif` and the output CIF `mtest.new` will be automatically opened. For a Unix OS the command will look like this:

```
tbx_exm > mtest.lst
```

3.2. Examine the test output

If you used the `Makefile` and the run produced no differences you are done. However, there may be differences between the expected output and your own run. One common case that should cause no concern arises from the way some systems present floating point numbers. Some systems present numbers in the form .123 or -.456, while others insert a leading zero with 0.123 and -0.456. This should not happen in any of the CIFs written by *CIFtbx2* ,since this behavior is controlled by the package, but the list outputs are not so constrained, and such differences may happen.

Here is what you may need to examine:

(g) to check that the test has been successful, compare the files that you have generated `test.lst` with the supplied `test.prt`, and `test.new` with `test.out`. They should be identical (except for the matter of leading zeros noted above).

(h) to check that the mmCIF test has been successful compare the files that you have generated `mtest.lst` with the supplied `mtest.prt`, and `mtest.new` with `mtest.out`. They should be identical, except as noted above.

Reporting Problems

If you have any problems installing *`CIFtbx2`* please contact:

Herbert J. Bernstein
em: yaya@bernstein-plus-sons.com,
ph: 1-516-286-1339, fax: 1-516-286-1999

or

Syd Hall
em: syd@crystal.uwa.edu.au
fx: 61(9)3801118

APPENDIX C

CYCLOPS2: a full *CIFtbx* application

CYCLOPS2 [Bernstein, Hall 97]is a new version of the program *CYCLOPS* [Hall 93] which is used, in conjunction with CIF dictionaries, to validate data names in an ASCII file which may contain CIF or non-CIF data, text documents or a program source. The new version is able to work with DDL1 or DDL2 dictionaries, the long data names of mmCIF dictionaries and with multiple dictionaries. *CYCLOPS2* is written incorporating the *CIFtbx2* [Hall, Bernstein 96] library of Fortran functions and is portable to a variety of platforms. *CYCLOPS2* is available along with *CIFtbx2*.

CYCLOPS is used primarily for checking spelling of CIF data names in documents and program sources against those in a CIF dictionary. It is also used to self-check a CIF dictionary for consistent definitions and cross-references.

Since the development of *CYCLOPS* in 1992 the CIF approach has been applied to new areas such macromolecular structural data. The mmCIF dictionary [Fitzgerald, Berman, Bourne, McMahon, Watenpaugh and Westbrook 96 and Bourne, Berman, McMahon, Watenpaugh and Westbrook 96] defines the structural parameters used in macromolecular studies and invokes an extended dictionary definition language referred to as DDL2 [Westbrook and Hall 95]. DDL2 involves stronger relational attributes than the DDL [Hall, Cook 95] used for the core crystallographic dictionary. *CYCLOPS2* can process dictionaries using either DDL or DDL2, as well as the longer data names of mmCIF, whereas the earlier *CYCLOPS* is limited to dictionaries using the simpler DDL attributes and 32 character names. *CYCLOPS2* uses the *CIFtbx2* library functions to perform the dictionary reading and CIF parsing operations.

Because of the increasing size of CIF dictionaries (the mmCIF dictionary alone adds nearly 1500 new data names not found in the core crystallographic dictionary) *CYCLOPS2* outputs the validation information quite differently from CYCLOPS . It lists the data names encountered in the validated file and dictionary files in three separate categories, the last one of which is optional: first output are the data names that are unrecognised (i.e. encountered in the validated file but not in the dictionary), second are the names that are present in both the validated file and the dictionary; and third, if requested, are the names encountered in the dictionary but not in the validated file. Aliased data names are also listed.

CYCLOPS2 overview

Files are read and written by *CYCLOPS2* as follows:

File a The text file to be validated is read from the *standard input device* (normally device 5). For Unix operating systems this is the file `stdin`; on other systems *CYCLOPS2* uses the file STARTEXT.

File b The dictionary file or files are identified in the input text file STARDICT. This file may itself be a DDL-conformant dictionary, or, if it begins with the characters "#DICT", may list the filenames of dictionaries to be entered, one per line. For Unix operating systems, this information may be provided directly on the command line.

File c The validation report is output to the *standard output device* (normally device 6). For Unix operating systems this is the file stdout; on other systems *CYCLOPS2* uses the file STARCHEK.

File d Messages are output to the *standard error device* stdout (normally device 0). For Unix operating systems this is the file stderr; on other systems *CYCLOPS2* uses device 6 and the file STARCHEK

The dictionary names are gathered into an array XDICT and then opened in the selected order in the following loop:

```
 do i = 1,kdict
 if(.not.dict_(XDICT(i)(1:max(1,lastnb(XDICT(i)))),prior))
*  call cerr(1,XDICT(i)(1:max(1,lastnb(XDICT(i))))//
*' not found',' ')
 MDICT(i)=NDICT
 DICTNM(i)=dicname_
 DICTVR(i)=dicver_
 enddo
```

This is the critical use of *CIFTBX2*. Both DDL and DDL2 dictionaries can be loaded with proper recognition of aliases between them. After that, CYCLOPS2 has the arrays of dictionary item names available to it.

The following procedure is used by *CYCLOPS2* to check data names:

1. Read STARDICT (see *File b* above) and, based on the first line, make a list of the dictionaries to be loaded. On Unix systems, add dictionaries specified on the command line as -d dicname to the list of dictionaries.

2. Load the dictionaries and store all data names. Any dictionary processing errors are reported to the error output file (see *File d* above).

3. Read the text file to be checked, parsing it line-by-line for identifiable data names (*i.e.*, text strings starting with the underscore character "_"). Data names are recognized provided the name begins with an underscore and the name is:

preceded by one of the characters <*blank*> <*tab*> , . ([{ < / \ | " ' : *

and

followed by one of the characters <*blank*> <*tab*> , .)] } > / \ | " ' - = ? ! ; : .

All alphabetic characters are converted to lower case. The scan stops at the comment character, "#", and goes to the next line.

4. On encountering a data name in step 3, search the stored dictionary names for a match. A match is attempted in one of three ways.

- If the data name is not preceded by the asterisk character "*", and it does not end with the underscore character "_", then search for an identical match.
- If the data name ends with the underscore character "_", then search for a match in the dictionary where the leading characters in the dictionary name are the same as all the characters in the data name found in the text. For example, the text `_atom_site.label_` would match the mmCIF dictionary entry `_atom_site.label_alt_id`
- If the data name is preceded by the asterisk character "*", then search for a match in the dictionary where the trailing characters in the dictionary name are the same as all the characters in the data name found in the text. The first match found in the dictionary is accepted. For example, the text `*_alt_id` would match `_atom_site.label_alt_id`, or, if that name had not been in the dictionary, `_struct_conn.ptnr1_label_alt_id`

If one of the searches succeeds, add the line number of the data name to a list attached to the dictionary name. Up to 19 line numbers are retained for each dictionary name (the first 10 matches and the last 9). If a data name has been misspelled it is likely to be caught at the next stage.

5. If no match is found, the unmatched data name is added to the a list of unmatched names, along with the appropriate line number. If a data name has been misspelled it will be caught at this step.

6. When the text file has been processed, output the validation report file (see *File d* above) containing the alphabetically sorted list unmatched names and line numbers, followed by the sorted list names from all dictionaries that are used within the text, followed, if requested, by the sorted list of names from all dictionaries that are not used within the text on the file. If a data name has an alias defined in the dictionaries, a warning about the existence of the alias is given. If more than one dictionary has been used, the source dictionary is identified for each data name. Example extracts from a validation output file are shown in Tables 1 and 2. A command line option is provided to suppress all but the list of unmatched names.

Table 1. *Sample output at the start of a validation output file. Note that the mmCIF dictionary, `cifdic.m96`, defines aliases for data names in the core dictionary, `cifdic.c94`.*

```
                CYCLOPS Check List
                ------------------


             Dictionary data names  = 1997
             New data names in text =    4
             [1]  Dictionary cifdic.c94 data names =   533
             [2]  Dictionary cifdic.m96 data names =  1464

 Data names NOT in Dictionary                          Line Numbers

 _blat1  . . . . . . . . . . . . . . . . . . . . . .     9    11    94    96
                                          181   183   290   296   302   308
                                          314
 _blat2  . . . . . . . . . . . . . . . . . . . . . .    13    15    98   100
                                          185   187   287   293   299   305
                                          311
 _dummy_test . . . . . . . . . . . . . . . . . . . .     5     7    90    92
                                          177   179   201
 _rubbish_here . . . . . . . . . . . . . . . . . . .   431


 [1]  Dictionary cifdic.c94
 [2]  Dictionary cifdic.m96
                                                       Line Numbers

 [2] _atom_site.calc_attached_atom . . . . . . . . .   413
 [1] = _atom_site_calc_attached_atom                   412
 [2] _atom_site.calc_flag  . . . . . . . . . . . . .   410
 [1] = _atom_site_calc_flag                            409
 [2] _atom_site.fract_x  . . . . . . . . . . . . . .    38    44    50   390
 [1] = _atom_site_fract_x                              389
 [2] _atom_site.fract_y  . . . . . . . . . . . . . .    39    45    51   394
 [1] = _atom_site_fract_y                              393
 [2] _atom_site.fract_z  . . . . . . . . . . . . . .    40    46    52   398
 [1] = _atom_site_fract_z                              397
 [2] _atom_site.id . . . . . . . . . . . . . . . . .    37    43    49   386
 [1] = _atom_site_label                                385
 [2] _atom_site.thermal_displace_type  . . . . . . .   406
 [1] = _atom_site_thermal_displace_type                405
 [2] _atom_site.type_symbol  . . . . . . . . . . . .   416   420   424   428
                                          434   438   442   450   454   458
                                          462
 [1] = _atom_site_type_symbol                          415   419   423   427
                                          433   437   441   449   453   457
                                          461
```

Table 2. *Sample output still later in a validation output file, showing the transition to unreferenced data names.*

```
[1] _symmetry_cell_setting  . . . . . .   319
[2] = _symmetry.cell_setting                320
[1] _symmetry_space_group_name_H-M  . . . .   323
[2] = _symmetry.space_group_name_H-M        324
[1] _symmetry_space_group_name_Hall . . . .   327   445
[2] = _symmetry.space_group_name_Hall       328   446
```

[1] Dictionary cifdic.c94
[2] Dictionary cifdic.m96

Names Not Referenced

```
[2] _atom_site.aniso_B[1][1]
[2] _atom_site.aniso_B[1][1]_esd
[2] _atom_site.aniso_B[1][2]
[2] _atom_site.aniso_B[1][2]_esd

 [... portion of output omitted ...]

[2] _atom_site.aniso_U[2][3]
[2] _atom_site.aniso_U[2][3]_esd
[2] _atom_site.aniso_U[3][3]
[2] _atom_site.aniso_U[3][3]_esd
[2] _atom_site.attached_hydrogens
[1] = _atom_site_attached_hydrogens
[2] _atom_site.auth_asym_id
[2] _atom_site.auth_atom_id
[2] _atom_site.auth_comp_id
[2] _atom_site.auth_seq_id
[2] _atom_site.B_iso_or_equiv
[1] = _atom_site_B_iso_or_equiv
[2] _atom_site.B_iso_or_equiv_esd
[2] _atom_site.cartn_x
[1] = _atom_site_Cartn_x
[2] _atom_site.cartn_x_esd
[2] _atom_site.cartn_y
[1] = _atom_site_Cartn_y
[2] _atom_site.cartn_y_esd

 [... remainder of output omitted ...]
```

Error Message Glossary

In addition to the error messages reported by the *CIFtbx2* library routines when processing dictionaries, *CYCLOPS2* can output the following error messages.

`Data name in text is > NUMCHAR chars` *<string>*
A non-fatal warning issued if the length of a data name in the validated file exceeds the preset value of `NUMCHAR`. Processing continues with a truncated name. The value of `NUMCHAR` should be changed in `ciftbx.sys` and `ciftbx.f` recompiled.

`Dictionary list empty`
The file STARDICT does not contain definitions or a list of dictionary filenames, and none were specified on the command line. See *File a* description above.

`Too many dictionaries`
More than 99 dictionaries have been added to the list of dictionaries. This is almost certainly due to an error in constructing the file STARDICT. In any case CYCLOPS2 cannot process more than 99 dictionaries. If many small dictionaries must be handled, break them up into groups of less than 100.

<dictionary name> **`not found`**
The dictionary file named in STARDICT or on the command line could not be opened.

`Data name table exceeded (Current max is NUMDICT)`
The combined number of data names in the dictionaries and the text is greater than the parameter `NUMDICT` defined in `ciftbx.sys`. Recompile with a larger value of `NUMDICT`.

Distribution

The latest version of this software is available as part of the *CIFtbx2* (see Appendix A).

CIFtbx2 distributed as the file `ciftbx.cshar`. C*YCLOPS2* is distributed as the companion file `cyclops.cshar`. There are several dictionaries that are useful for validation. In general the latest versions can be traced by starting at the IUCr CIF web page at:

`http://www.iucr.org/iucr-top/cif/`

or one of its mirror sites (see Appendix A).

APPENDIX D

Syntax of a STAR File

Let us look more carefully at the syntax of a STAR file. A "String" is one or more printable ANSI/ISO characters, including blank or tab. A "starString" is a string of one or more printable characters not beginning with an underscore, hash-mark or dollar sign, not containing a blank or tab, and not matching one of the special words listed above. White space consists of blanks, tabs or comments. Except when identifying names with underscore, data_, save_ or a dollar sign, white space must always separate the tokens in STAR, which means that the same white space does double duty ending one syntax element and starting another. To keep the syntax descriptions as simple as possible, we indicate the mandatory presence of white space (or, in the case of a semicolon, of an end of line) which is part of the context of a syntactic element, but which may be part of another syntactic element by shading the background.

$$\langle whiteSpace \rangle = \{\langle blank \rangle \mid \langle tab \rangle \mid \{\{\langle blank \rangle \mid \langle tab \rangle\}\#\{\langle String \rangle\}_0^1\}_0^1 \langle eol \rangle\}_1^\infty$$

We make a STAR file out of a series of blocks separated by whitespace:

$$\langle STARfile \rangle = \{\langle whiteSpace \rangle\}_0^1 \{\langle block \rangle \langle whiteSpace \rangle\}_0^\infty$$

In a STAR file, a block may be an un-named `global_` block or a named `data_` block:

$$\langle block \rangle = \langle global\_Block \rangle \mid \langle data\_Block \rangle$$
$$\langle global\_Block \rangle = \langle global\_ \rangle \langle whiteSpace \rangle \langle blockBody \rangle$$
$$\langle data\_Block \rangle = \langle data\_ \rangle \langle name \rangle \langle whiteSpace \rangle \langle blockBody \rangle$$
$$\langle global\_ \rangle = \{g \mid G\}\{l \mid L\}\{o \mid O\}\{b \mid B\}\{a \mid A\}\{l \mid L\}\_$$
$$\langle data\_ \rangle = \{d \mid D\}\{a \mid A\}\{t \mid T\}\{a \mid A\}\_$$

In a CIF there are no `global_` blocks, but they are used in dictionaries. The body of a block consists of white space separated tag-value pairs, loops and `save_` frames. `Save_` frames are not used in CIF, but they are used in the mmCIF dictionary.

$$\langle blockBody \rangle =$$
$$\{\{\langle tag \rangle \langle whiteSpace \rangle \langle value \rangle \mid \langle loop \rangle \mid \langle save\_Frame \rangle\} \langle whiteSpace \rangle\}_0^\infty$$

A `save_` frame is very similar to a `data_` block, but it cannot contain an embedded `save_` frame, and it must be terminated by a `save_`.

⟨*save_Frame*⟩ = ⟨*save_*⟩⟨*name*⟩ ⟨*whiteSpace*⟩ ⟨*frameBody*⟩ ⟨*whiteSpace*⟩ ⟨*save_*⟩

⟨*frameBody*⟩ =

$$\left\{\left\{\langle tag\rangle\ \langle whiteSpace\rangle\ \langle value\rangle \mid \langle loop\rangle\right\}\ \langle whiteSpace\rangle\right\}_{0}^{\infty}$$

⟨*save_*⟩ = {s | S}{a | A}{v | V}{e | E}_

A loop consists of `loop_`, `whitespace`, the header for the loop giving the tags that label the columns of the table being specified, and then the body of the loop giving one or more rows of values to be associated with the tags in the header. STAR permits nested loops, with appropriately aligned sublists in the header and body delimited by `stop_` CIF does not permit any nesting, and `stop_` is not used. STAR also permits a value to be a "frame code", a reference to a `save_` frame indicated by a dollar sign before the name of the `save_` frame. The feature is not used in CIF. The values are starStrings, or quoted strings. Note that the initial end of line (eol) is part of the context of the initial delimiter and part of the prior white space.

⟨*loop*⟩ = ⟨*loop_*⟩ ⟨*whiteSpace*⟩ ⟨*loopHeader*⟩ ⟨*loopBody*⟩

⟨*loopHeader*⟩ =

$$\left\{\left\{\langle tag\rangle \mid \langle loop\_\rangle\ \langle whiteSpace\rangle\langle loopHeader\rangle\langle whiteSpace\rangle\langle stop\_\rangle\right\}\ \langle whiteSpace\rangle\right\}_{1}^{\infty}$$

⟨*loopBody*⟩ = $\left\{\left\{\langle value\rangle \mid \langle stop\_\rangle\right\}\ \langle whiteSpace\rangle\right\}_{1}^{\infty}$

⟨*value*⟩ =

$$\left\{\langle starString\rangle \mid '\langle String\rangle' \mid "\langle String\rangle" \mid \langle eol\rangle;\left\{\left\{\langle String\rangle\right\}_{0}^{1}\langle eol\rangle\right\}_{1}^{\infty}; \mid \$\langle name\rangle\right\}$$

⟨*loop_*⟩ = {l | L}{o | O}{o | O}{p | P}_

⟨*stop_*⟩ = {s | S}{t | T}{o | O}{p | P}_

APPENDIX E

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, use of the `ENDDO` statement, the names of routines used internally by the package, use of names longer than six characters and 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 processing whitespace.

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.

The variables declared in `ciftbx.cmn` are organized into three labeled common blocks:

```
      common /tbuc/ strg_, bloc_, file_, type_, dictype_,
     *  diccat_, dicname_, dicver_, tagname_, quote_,
     *  pquote_, tbxver_
      common /tbui/ list_, long_, longf_, line_, esdlim_,
     *  recn_, precn_, posnam_, posval_, posdec_,
     *  posend_, pposnam_, pposval_, pposdec_, pposend_,
     *  recbeg_, recend_
      common /tbul/ loop_, text_, align_, save_, saveo_,
     *  aliaso_, alias_, tabl_, tabx_, ptabx_, nblank_,
     *  nblanko_, glob_, globo_, decp_, pdecp_, lzero_,
     *  plzero_, append_
```

The blocks with labels `/tbuc/`, `/tbui/`, and `/tbul/` are for variables of type character, integer and logical respectively. The additional internal variables declared in ciftbx.sys are similarly organized into labeled common blocks, `/tbxc/`, `/tbxi/`, `/tbxdp/`, `/tbxr/`, and `/tbxl/` for variables of type character, integer, double precision, real and logical, respectively. Portability is enhanced not mixing common for variables of different types.

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.

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:

NUMCHAR Maximum number of characters in data names (default 48)

MAXBUF Maximum number of characters in a line (default 200)

NUMPAGE Number of memory resident pages (default 10)

NUMCPP Number of characters per page (default 16384)

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)

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 size of the data transfers to and from that file. Smaller values will reduce the demand for memory at the expense of slower execution.

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

```
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.

The system common, ciftbx.sys contains

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

but not the type declarations in `ciftbx.cmf` in order to avoid an excessive number warning messages about unreferenced variables produced by some compilers.

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.

Names of Internal Routines

The following routines are used internally by *CIFtbx* version 2.6. If these names are needed for other routines, then changes in the library will be needed to avoid conflicts.

Variable Initialization:

```
      block data
```

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

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.

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.

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.

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 esd. The subroutine putnum converts a number and esd to an output string.

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.

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.

Use of the Underscore Character

All the externally accessible *CIFtbx* commands and variable terminate with the underscore character. This works well on most systems, but can cause occasional problems, since 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 names was changed to a distinctive character pattern (e.g. "CIF" or "qq"), but caution is needed, since there are many character strings 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'
```

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.

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 were 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 out of the order 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.

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`.

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. An application needing maximum speed and not needing the information and wishing to conserve space on disk might wish to modify *CIFtbx* to remove all comments and compress all separating white space to single blanks or line terminators in an initial sweep.

Bibilography

[Allen 96] "The Protein Data Bank and the Cambridge Structural Database: Inter-relationships in Construction and Utilization", 17th IUCr Congress and General Assembly, Seattle, Washington, USA, 8-17 August 1996, Abstract S0232, 1996.

[Allen, Barnard, Cook, Hall 95] F. H. Allen, J. M. Barnard, A. F. P. Cook. and S. R. Hall, "The Molecular Information File (MIF): Initial Specifications", *J. Chem. Inform. Comp. Sci.*, Vol. 35, 1995, 412-427.

[Berman, Westbrook 93] H. M. Berman and J. Westbrook, "How the NDB uses CIF", in "Proceedings of the First Macromolecular Crystallographic Information (CIF) Tools Workshop", P. E. Bourne, ed., October 15-18, 1993, Tarrytown, NY, Howard Hughes Medical Institute, Columbia University, NY, 1993, pp. 63-66.

[Bernstein 97] H. J. Bernstein, "Software Engineering", Encyclopedia of Applied Physics, Vol. 18, VCH Publishers, Inc., 1997, pp 305-340.

[Bernstein 97a] H. J. Bernstein, "*CIFtbx* Applications. cif2cif: A program to copy CIFs", in prepapration.

[Bernstein, Andrews 96] H. J. Bernstein and L. C. Andrews, "Software for Lattice Identification in G-6," invited presentation for Seattle IUCr meeting, 7-17 August 1996.

[Bernstein, Bernstein 96] F. C. Bernstein and H. J. Bernstein, "Translating mmCIF Data into PDB Entries", invited presentation for CIF workshop at Seattle IUCr meeting, 7-17 August 1996. software available at http://www.bernstein-plus-sons.com//software/cif2pdb

[Bernstein, Bernstein, Bourne 98] H. J. Bernstein, F. C. Bernstein and P. E. Bourne, "CIF Applications. pdb2cif: Translating PDB Entries into mmCIF Format", *J. Appl. Cryst.*, accepted, software available at http://ndbserver.rutgers.edu/NDB/mmcif/software

[Bernstein, Hall 96] H. J. Bernstein and S. R. Hall, "CIF Applications. *CYCLOPS2:* Extending the Validation of CIF Data Names," *J. Appl. Cryst.*, to appear, software available at http://ndbserver.rutgers.edu/NDB/mmcif/software

[Bernstein, Koetzle, Williams, Meyer, Brice, Rodgers, Kennard, Shimanouchi, Tasumi 77] F. C. Bernstein, T. F. Koetzle, G. J. B. Williams, E. F. Meyer Jr., M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi and M. Tasumi, "The Protein Data Bank: A Computer-Based Archival File for Macromolecular Structures", *J. Molec. Biol.* Vol. 112,1977, pp. 535-542. Also published in *Eur. J. Biochem. Vol.* 80, 1977, pp. 319-324, *Arch. Biochem. Biophys.* Vol. 185, 1978, pp. 584-591.

[Bourne 93] P.E. Bourne, ed., "Proceedings of the First Macromolecular Crystallographic Information (CIF) Tools Workshop", October 15-18, 1993,

Tarrytown, NY, Howard Hughes Medical Institute, Columbia University, NY, 1993, 127 pp.

[Bourne, Berman, McMahon,. Watenpaugh, Westbrook, Fitzgerald 96] P. E. Bourne, H. M. Berman, B. McMahon,. K. D.. Watenpaugh, J. Westbrook, and P. M. D. Fitzgerald, "The Macromolecular Crystallographic Information File (mmCIF)", Methods in Enzymology, 1996, submitted.

[Bourne, Bernstein, Bernstein 96] P. E., Bourne, H. J. Bernstein and F. C. Bernstein, "Translating PDB Entries into mmCIF", invited presentation for CIF workshop at Seattle IUCr meeting, 7-17 August 1996.

[Fitzgerald, Berman, Bourne, McMahon, Watenpaugh, Westbrook 96] P. M. D. Fitzgerald, H. M. Berman, P. E. Bourne, B. McMahon, K. D.Watenpaugh, and J. Westbrook, "The MMCIF Dictionary: Community Review and Final Approval," 17th IUCr Congress and General Assembly, Seattle, Washington, USA, 8-17 August 1996, Abstract E1226, 1996. Version 0.9.01 available from http://ndbserver.rutgers.edu

[Gelbin, Westbrook, Berman 95] A. Gelbin, J. Westbrook, H. Berman, "mmCIF Data Set DDF040", 1995, available from http://ndbserver.rutgers.edu derived from G. A. Leonard, T. W. Hambley, K. McAuley-Hecht, T. Brown, W. N. Hunter, "Anthracycline-DNA Interactions at Unfavourable Base-Pair Base-Pair Triplet-Binding Sites: Structures of d(CGGCCG)/Daunomycin and d(TGGCCA)/Adriamycin Complexes", *Acta Cryst.*, Vol. D49, 458 ff., 1993.

[Hall 91] S. R. Hall, "The STAR File: A New Format for Electronic Data Transfer and Archiving", J Chem Inform Comp Sci., Vol. 31, 1991, pp. 326-333.

[Hall 93a] S. R. Hall, "CIF Appl. II. CIFIO: for CIF Input/Output in the Xtal System", *J. Appl. Cryst.*, Vol. 26, 1993, pp. 474-479.

[Hall 93b] S. R. Hall, "CIF Appl. III. CYCLOPS: for validating CIF Data Names", *J. Appl. Cryst.*, Vol. 26, 1993, pp. 480-481.

[Hall 93c] S. R. Hall, "CIF Appl. IV. *CIFtbx* a Toolbox for Manipulating CIF's", *J. Appl. Cryst.*, Vol. 26, 1993, pp. 482-494.

[Hall, Allen, Brown 91] S. R. Hall, F. H. Allen and I. D. Brown, "The Crystallographic Information File (CIF): A New Standard Archive File for Crystallography", *Acta Cryst.*, Vol. A47, 1991, pp. 655-685.

[Hall, Bernstein 96] S. R. Hall and H. J. Bernstein, "CIF Applications. *CIFtbx2:* Extended Tool Box for Manipulating CIFs", *J. Appl. Cryst.*, Vol. 29, 1996, pp 598-603.

[Hall, King, Stewart 95] S. R. Hall, G. D. S. King and J. M. Stewart, "Xtal 3.4 Users Manual. Report", University of Western Australia, 1995.

[Hall, Cook 95] S. R. Hall and A. P. F. Cook, "Data Definition Language for STAR File Dictionaries", *J. Chem. Inform. Comp. Sci.*, Vol. R35, 1995, pp. 819-825.

[Hall, Sievers 93] S. R. Hall & R. Sievers, "CIF Appl. I. QUASAR: for extracting CIF data", *J. Appl. Cryst.*, Vol. 26, 1993, pp. 469-473.

[Hall, Spadaccini 94] S. R. Hall and N. Spadaccini, "The STAR File: Detailed Specifications", *J. Chem. Inform. Comp. Sci.*, Vol. 34, 1994, pp. 505-508.

[Holland 96] A. J. Holland, "CCP14 in Powder and Single-Crystal Diffraction", 17th IUCr Congress and General Assembly, Seattle, Washington, USA, 8-17 August 1996, Abstract E0119, 1996. See http://www.dl.ac.uk/CCP/CCP14

[IUCr 95] "A Guide to CIF for Authors", International Union of Crystallography, Chester, England, 1995, 16 pp.

[IUCr 96] "Notes for Authors", *Acta Cryst.* , Vol. C52, 1996, pp. 265-278.

[Keller 96] P. A. Keller, "A mmCIF Toolbox for CCP4 Applications", presentation for CIF workshop at Seattle IUCr meeting, 7-17 August 1996, abstract E0726, 1996.

[Knuth 73] D. E. Knuth,KNUTH, D. E.. (1973) "The Art of Computer Programming, Volume 3/Sorting and Searching", Addison Wesley Publishing Company, Reading MA,1973, 722 pp.

[Knuth 84] D. E. Knuth, "The TEXbook", Addison Wesley Publishing Company, Reading MA, 1994, 481 pp.

[McMahon 93a] B. McMahon, "CIF Software at Chester", in "Proceedings of the First Macromolecular Crystallographic Information (CIF) Tools Workshop", P.E. Bourne, ed., October 15-18, 1993, Tarrytown, NY, Howard Hughes Medical Institute, Columbia University, NY, 1993, pp. 55-56.

[McMahon 93b] B. McMahon, "How does my CIF become a printed paper?", *Acta Cryst.*, Vol. C49, 1993, pp. 418-423.

[McMahon 95] B. McMahon, "A Brief History of the DDL", COMCIFS, 1995. http://www.iucr.ac.uk/iucr-top/cif/ddlhist.html

[PDB 96] "Protein Data Bank Contents Guide: Atomic Coordinate Entry Format Description", Version 2.1 (draft), October 25, 1996, Protein Data Bank, Upton, NY, 1996, 156 pp.

[Sayle 94] R. Sayle, "Rasmol 2.5 Molecular Graphics Visualization Tool," BioMolecular Structures Group, Glaxo Research & Development, Greenford, Middlesex, UK., October 1994.

[Sheldrick 95] G. M. Sheldrick, "PDB entry 1CTJ, Crystal Structure of Cytochrome c6," Protein Data Bank, 8 August 1995, from C. Frazao, C. M. Soares, M. A.

Carrondo, E. Pohl, Z. Dauter, K. S. Wilson, M. Hervas, J. A. Navarro, M. A. De La Rosa, G. M. Sheldrick, "Ab Initio Determination of the Crystal Structure of Cytochrome c6, Comparison with Plastocyanin," *Structure (London)* Vol. 3, 1159-1169, 1995.

[Spadaccini, Hall 94] N. Spadaccini & S. R. Hall, "Star_Base: Accessing STAR File Data", *J. Chem. Inform. Comp. Sci.* Vol. 34, 1994, pp. 509-516.

[Stampf 94] D. R. Stampf, "ZINC -- Galvanizing CIF to work with UNIX," Protein Data Bank, Upton, NY 1994.

[Stampf *et al.* 96] D. R. Stampf, E. E. Abola, N. O. Manning, D. Xue, J. L. Sussman, "AutoDep - Facilitating Deposition to the Protein Data Bank through the New Web-Based Submission Form", 17th IUCr Congress and General Assembly, Seattle, Washington, USA, 8-17 August 1996, Abstract E1205.

[Toby 97] B. H. Toby, "Powder CIF Dictionary," version 0.996, available from http://www.iucr.ac.uk/iucr-top/CIF

[Westbrook, Hall 95] J. Westbrook and S. R. Hall, "A Dictionary Description Language for Macromolecular Structure, Draft DDL V 2.1.0", IUCr COMCIFS, Chester, England, 1995.

[Westbrook, Hsieh, Fitzgerald 97] J. D. Westbrook, S.-H. Hsieh and P. M. D. Fitzgerald, "CIFLIB: An Application Program Interface to CIF Dictionaries and Data Files", *J. Appl. Cryst.*, Vol. 30, 1997, pp. 79-83.

Index