CCIF: the mmCIF toolbox for CCP4. A Programmer's Guide

Peter A. Keller ⟨keller@ebi.ac.uk⟩

Release 0.3 (September 8, 1999)

Contents

Introduction.

I'll think of something pithy to put in here in due course

Part I

Introduction to CIF's.

1 How the CIF specifiction works.

1.1 Analysing text files.

A CIF is a text file, which is designed to be machine-readable, and also human-readable, by people who are knowledgeable about the specification. (In this sense, CIF's are similar to source code written in programming languages such as Fortran and C.) The fact that human beings have very different ways of analysing and assimilating information from 'ordinary' computer programs, has important consequences for how the rules for such files are specified. In general, a computer program cannot just glance at a the contents of an arbitrary text file, form an overall impression of its contents, and then skip to the relevant bit, as a human being would. The program has to read the file as a stream of characters, and interpret them according to a set of rules, before it has any access to the information which is contained in the file.

This process falls, broadly, into three stages. The first stage is known as 'lexical analysis', and involves identifying important groups of bytes, which are known as 'tokens'. For example, in a fragment of a CIF such as:

```
data_EXAMPLE

   _symmetry.entry_id                    EXAMPLE
   _symmetry.space_group_name_H-M       'P 21 21 21   '
```

we would start by noting that the input begins with a data block heading, which is identified by its initial five bytes '`d`', '`a`', '`t`', '`a`', '`_`', and which continues up to the next space character. We then have a data name (which begins with a '_'), a non-quoted text string, another data name, and a single-quoted text string.

The input into this process consists of a byte stream of about 120 characters, and the output is a 'token stream' consisting of the five tokens `<data_heading>`,

<data_name>, <non_quoted_text_string>, <data_name>, and <single_quoted_text_string>. In this process, lexically unimportant characters are discarded, for example spaces which separate tokens (although the spaces in the quoted string are significant, and so are retained).

The next step in the process of interpreting the information, is known as 'syntactical [or grammatical] analysis'. At this stage, the token stream is analysed (or parsed), and checked for errors. For example, a <data_name> may not be followed by a <data_heading> in the token stream, and such a sequence should cause an error condition to occur. A <data_name> followed by a <single_quoted_text_string> is allowed, and means that the contents of the text is assigned to the data name.

The third stage is known as 'semantic analysis', and consists of actually interpreting the results of the previous stage. In our example above, it is during this process that we note that the data name *_symmetry.space_group_name_H-M* refers to the space group name in Hermann-Mauguin notation, and that in this CIF, it is defined to be $P2_12_12_1$.

In practice, there can be some flexibility about where syntactical analysis takes over from lexical analysis — the discussion above reflects the way it has been implemented in CCIF. Any application which is written to accept its input as a sequence of characters, is implementing these three stages of analysis (at least implicitly). This applies even to the simplest 'jiffy' program.

1.2 How this is applied to CIF's.

It is worthwile being clear about these three stages of analysis, since otherwise there is scope for misinterpreting the contents of the CIF dictionaries, or becoming confused about what the CIF standard can and cannot specify. This is because the lexical and syntactical rules which CIF's and CIF dictionaries follow are part of a standard known as STAR[1] which exists independently of CIF. This means that every CIF (and CIF dictionary) must be a valid STAR file, and nothing can be introduced into the CIF standard which would allow a CIF to be invalid according to the STAR rules[2]. This has important consequences for the construction of non-quoted text strings (see below, section 1.6).

With this in mind, the CIF standard can be seen as a set of restrictions on the STAR syntax (discussed by Hall *et al.*[3]) together with the specification of the semantic rules

[1] Self-defining Text Archive and Retrieval; see 'The STAR File: Detailed Specifications', S.R. Hall, N. Spadaccini, *J. Chem. Inf. Comput. Sci*, **1994**, *34*, 505-508

[2] However, the CIF standard can, and does, forbid certain constructions which are allowed in STAR. In principle, these restrictions could be removed in future versions of the CIF standard.

[3] 'The Crystallographic Information File (CIF): a new Standard Archive File for Crystallography', S.R. Hall, F.H. Allen, I.D. Brown, *Acta Cryst.* **1991**, *A47*, 655-685. It should be noted, though, that this paper introduces the 'core CIF dictionary', and some aspects of the discussion need to be qualified when applied to the mmCIF standard. In particular, the restriction of 32 characters on data names does not apply to mmCIF.

which CIF’s follow[4]. In other words, the CIF standard specifies how crystallographic and structural information is expressed by the use of a subset of the STAR syntax, and how to interpret it.

1.3 The structure of CIF’s, and the role of CIF dictionaries.

The highest level of grouping of data in a CIF is the data block. This begins with a data block header (see the example above), and continues to the next data block header (or end of file). Each data block must be uniquely identified by the characters which follow the ‘*data_*’ part. Data defined in a data block is restricted in scope to that block, i.e. it cannot be referred to from outside the data block. This is a semantic rule, but it is part of the STAR specification, so the CIF rules may not specify otherwise. This rule, in combination with the restrictions which CIF places on STAR, mean that each data block in a CIF is self-contained, and independent of anything in the CIF which occurs outside of that data block.

The data block contains assignments to data names, either as <data_name> <data_value> pairs, or in a loop structure, (which works like a table). It is the function of the CIF dictionary to specify the properties of particular data names (including, crucially, the relationships between them), and the permitted forms of data values which may be assigned to them. The various items are grouped together in categories, which are defined in the dictionary. (In mmCIF, the part of the data name which occurs in between the initial ‘_’ and the ‘.’ is the name of its category.) The properties of the categories themselves are also defined in the dictionary[5].

1.4 More about CIF categories

The concept of categories is important in understanding how loops work. The basic strucure of a loop is defined in the STAR specification; the CIF standard exploits this construction to represent the contents of the categories of data which are defined in the dictionaries. All the data names in the header of a loop structure must come from the same CIF category. If there is more than one ‘packet’ (i.e. row) in the loop, each packet must be uniquely identified, in order to distinguish it from the others. This is done by specifying in the dictionary that one or more items in a category is/are ‘key’ items. No loop packet may have a set of key values[6] which are identical to those of another loop packet. As an example, in the *REFLN* category, the key items are those which specify the Miller indices $(h\,k\,l)$.

[4] Of the semantic rules which are specified by STAR, there is one which is important for CIF’s. This is mentioned in the next section.

[5] This begs the question as to how one knows which properties of items can be legitimately defined in a CIF dictionary. The answer is the CIF DDL (Dictionary Definition Language), aka the schema. Discussion of DDL’s seems to confuse almost everyone, so no more will be said about them here, except to point out that a family of CIF dictionaries may exist, which all follow the same DDL. An example of this is the mmCIF dictionary, and the forthcoming NMRIF dictionary for NMR protein structure determination.

[6] A set of key values is sometimes referred to in database methodology as a ‘tuple’.

```
loop_
_atom_site.label_seq_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
. HETA CL CL     CL .  336  .  25.517  16.297  -6.059
      1.00 25.69  .    4   1683
. HETA CL CL     CL .  343  .   8.030  39.142  10.823
      1.00 27.91  .    4   1690
# other atoms deleted
```

Figure 1: The choride ions from structure 5HVP.

The various categories are connected by 'child-parent' links. This allows information which is specified in one category to be selectively applied to certain loop packets in another category. For example, the data item *_atom_site.type_symbol* is a child of (or 'pointer to') the data item *_atom_type.symbol*. These items both contain the type (usually a simple element symbol) of an atom. The former occurs in the category *ATOM_SITE*, which contains information on each atom in the structure (coordinates, occupancies, etc.). The latter occurs in the category *ATOM_TYPE*, which defines properties of the various types of atoms which occur in the structure (scattering properties, atomic radii, etc.). Consider a case where there are several chloride ions in a structure (this example is taken from the PDB entry 5HVP). The *ATOM_SITE* category might look like that in figure 1.

The first 'CL' in the loop packets for each of these two atoms, are the children of the 'CL' in the *ATOM_TYPE* example shown in figure 2. This indicates that the scattering factors and oxidation number given for the chloride ion type applies to both of the choride ions which are specified the *ATOM_SITE* category.

A CIF category may be defined partly as a sequence of <data_name>

```
loop_
_atom_type.symbol
_atom_type.oxidation_number
_atom_type.scat_Cromer_Mann_a1
_atom_type.scat_Cromer_Mann_a2
_atom_type.scat_Cromer_Mann_a3
_atom_type.scat_Cromer_Mann_a4
_atom_type.scat_Cromer_Mann_b1
_atom_type.scat_Cromer_Mann_b2
_atom_type.scat_Cromer_Mann_b3
_atom_type.scat_Cromer_Mann_b4
_atom_type.scat_Cromer_Mann_c
  CL -1  18.2915    0.006600  7.20840    1.17170
    6.53370  19.5424    2.33860   60.4486    -16.378
# information on other atom types deleted
```

Figure 2: Scattering parameters for choride ions.

<data_value> pairs, and partly as looped data. In that case, the values specified outside the loop, apply to each loop packet. Clearly, at least one of the key items for the category must occur within the looped data, otherwise the different loop packets will not be unique. In principle, the looped data may be specified in more than one loop per data block[7]. I consider that this kind of structure is to be avoided, since there is too much scope for mutually incompatible interpretations, and misunderstanding. The more sophisticated use of categories in the mmCIF standard should make such multiple loops unnecessary, in any case. CCIF imposes the restriction that any data name can only occur once in each data block, so no more than one loop can be used in the definition of a category.

1.5 The specification of null and unknown values.

The STAR specification does not make explicit provision for null and unknown data values. However, by convention an isolated '`.`' represents a null value, and an isolated '`?`' an unknown value[8]. The distinction between the two can be rather subtle, and has been the subject of some debate. Essentially, a null value represents a value of a data item which is not applicable to, or meaningful in, the context in which it occurs. An example would be the value of *_atom_site.label_alt_id* for an atom which has been modelled in only one conformation.

[7]This is mentioned in the DDL for the core dictionary, although without being clear on which of several possibilities are allowed. The DDL which forms the basis of mmCIF is silent on this question.

[8]CCIF interprets these two characters as null and unknown when they are quoted as well — this is not strictly correct, but the practice of quoting them in CIF's seems to be widespread.

An unknown value represents one which would be meaningful, if the value were known. Unmeasured data would normally be represented in a CIF as unknown. CCIF treats data values which are not specified in a data block at all as formally equivalent to unknown values. This derives from the comment in the STAR paper[1] (§7(e)) that if the value of an item is not specified, it is unknown.

1.6 Privileged STAR constructs, and non-quoted text.

There are certain tokens defined in the STAR syntax, which are not used in CIF's. Nevertheless, the CIF programmer needs to be aware of them, to avoid accidentally using them for non-quoted text. The tokens are: `global_`, anything which starts with `save_`, including just `save_`, and anything which starts with `$`. Also, the token `stop_`, while allowed in CIF's, is in practice, rarely seen. Data which might seem to be non-quoted text strings, but which are, in fact one of these constructs, will cause the CIF to break the STAR rules (or, at the very least, will drastically change the interpretation of the data). If asked to output a data value which is an invalid non-quoted text string, CCIF will attempt all quoting allowed by the CIF dictionary specification for the item in an attempt to make it legal STAR data.

Part II

Using CCIF.

2 Basic Concepts

Standard FORTRAN77 lacks a syntax for representing or manipulating memory addresses, and working with complex data structures can be difficult. For this reason, manipulating data with a flexible structure can be a bit like reading a map with a microscope — it is hard work to get an overview of the data. CCIF tries to shield the programmer from the worst consequences of this, by maintaining information on the data and what the application program is doing internally, and only providing tightly controlled access to the data structures and their contents. Making best use of this requires some awareness of what is happening 'under the bonnet', and there are a number of concepts which must be clearly understood.

2.1 Logical names and input and output files

CCIF, for various reasons, handles its own input and output, and maintains its own list of open files. Nevertheless, logical names are basically the same as in standard CCP4 programs. They can be connected to real files under Unix by setting environment variables, and under VMS by using the `DEFINE` command. Programs which call the

CCPFYP subroutine can use the normal CCP4 mechanism of logical name–filename pairs on the command line. Within application programs, CCIF files are always referred to by their logical names, never by the actual filenames.

One area of difference arises from the fact that a CIF can contain multiple data blocks, and that a data block can contain arbitrary categories of information (only limited by what is in the CIF dictionary being used). This means that where a CCP4 application would traditionally access two or more files, all the sets of information may in fact be present in just one CIF. CCIF allows for this, by permitting different logical names to point to the same file. So, for example, if you want to use two sets of coordinates, which are in different data blocks in one CIF, you could do:

```
lsqkab WORKCD coordset.cif REFCD coordset.cif
```

Another situation might be where two different categories of input (e.g. coordinates and reflections) are actually in the same data block. In such a case, the following could be done:

```
sfall XYZIN proj.cif HKLIN proj.cif
```

These assume that `lsqkab` and `sfall` have been converted, so that they accept keywords to identify the relevant data blocks within the CIF. See section **??** for ideas on how to do this. The intention is that programs which use CCIF should be written in such a way that this is all transparent to the user of the application.

This also raises the possibility that the same file can legitimately be referred to in different ways (e.g. with or without pathnames, or *via* symbolic links). CCIF copes with this under Unix by comparing i-node numbers (these are the numbers you get with the `ls -i` command) and device ID's to determine whether two files are in fact the same. Under VMS, the file ID's are compared (these are the numbers which are obtained with the `DIR/FULL` command), unless DECnet specifications are used, in which case the filenames are fully expanded and compared as strings.

Most CCP4 programs make a distinction between input and output files, in that a program will not alter input files in any way, instead writing changed or new information out to a new file. This turns out to be useful practice for CIF's, since the information in a file may be accessed in an order which bears no relation to its contents. If an application program is allowed to directly alter such information, the program may be modifying its own input in subtle ways. Accordingly, CCIF hardens this convention into a rule: if a CIF is opened for input, on any attempt to alter its contents, CCIF will throw an error and terminate the program.

CCIF can cope with a maximum of 10 logical names being in use at any one time — this is set with the (slightly mis-named) 'C' preprocessor symbol `MAX_CIF_FILES`.

2.2 Specifying data within a file

Obviously, a CIF can contain an arbitrary number of data blocks. When a CIF is read or created by a CCIF application, a unique number (the block id) is assigned to each data block within that file (starting from 0), and it is necessary to specify this number in order to access or modify any data within that block. When a new data block is created in an output CIF, it is also given a new block id. The block id can be found in two ways, either by specifying the logical name and the data block name, or by specifying the logical name and the sequential number of the block within the file to which it refers. (This is done with the CCIF routines *CCIF_BLOCK_BY_NAME* and *CCIF_BLOCK_BY_INDEX* respectively. See sections 5.1 and 5.2.) In the case of an output file, *CCIF_BLOCK_BY_NAME* can be made to create a new data block.

To actually access data, it is necessary to initialise a 'context' on the appropriate category of data. As far as the application program is concerned, a context is just an integer (≥ 0) which is assigned when the context is set up. Internally, though, CCIF maintains information on the location of data in the CIF, and the context is used to provide rapid access to it. In the case of looped data, it points to the current packet. Initialising a context is done with the CCIF routine *CCIF_SETUP_CONTEXT* (see section 5.3), which needs the block id and the name of the category as input. If the logical name has been opened for output, this routine can create the structures for inserting a new category of information into the file.

Once a context is set up for a particular category, individual data items can be retrieved or output, by specifying the appropriate data name and context. In the case of output, data names which are not already present in the file can be inserted, along with their values. Setting up multiple contexts on the same category of data simultaneously is permitted. Where the context has been opened on looped data, it can be advanced one packet at a time[9]. Changes made to a data category through one context are reflected immediately on all contexts open on that category, except where a data manipulation routine has already returned `end_of_context` in response to `advance_and_hold` (see section 6).

2.3 Sorting and searching.

The CIF standard allows the packets of a loop to occur in any order[10]. CCIF allows applications to specify a sort order for a category, and to access the loop packets in that order. Applications may also search for a loop packet which contains a particular combination of values, as long as an appropriate sort has been set up on the category.

[9] If no sorting has been applied to the context, the packets will be retrieved in the order in which they occur in the file. See the next section, and section 7, for information on how to access loop packets in sorted order.

[10] In this context, it may be noted that the PDB standard Version 2.0, (February 21, 1996) specifies that where a standard residue side chain is modified by adding atoms, the new atoms occur after the end of the chain in which the residue occurs, not immediately after the other atoms in the same residue. This implies that atoms with a particular chain id and residue id may occur at more than one location in the file.

Handling sorting is very similar conceptually to using contexts. A sorting structure is set up on a category within a data block, with a call to *CCIF_SETUP_SORT*. Among other things, the structure maintains a list of data items (a 'sort list'), which specifies which values are compared, and in what order, when deciding on the relative order of loop packets. This routine returns an index, which is used by the application to refer to the sort in future calls to CCIF routines which make use of it.

The sort list is set up by making repeated calls to *CCIF_ADD_TO_SORT*, and the sort itself is invoked with a call to *CCIF_DO_SORT*. A previously set up context can then be made to access a category in sorted order, with a call to *CCIF_SORT_CONTEXT*. For more details, see the descriptions of these functions in section 7. Note that this does not change the underlying order of items within a looped data category, merely the order in which they are accessed. More than one sort order may be active simultaneously on a category, and be accessed through different contexts.

Sorting is done in ascending order, and individual values are compared according to the item's *_item_type_list.primitive_code*: numerical for all numerical types, and case-senstitive or case-insenstive ASCII order for the primitive types 'char' and 'uchar' respectively. Null values (`'.'`) come first in the sorting order, and Unknown values (`'?'` or absent) come last. If a loop packet's value is changed using one of the *CCIF_PUT_** routines, and the item concerned is involved in one or more sorts, the loop packet will go to its new location immediately, taking any contexts which are pointing to that loop packet with it[11]. If the *_item_type_list.construct* which applies to an item, allows the item to have a multi-line text value, the item may not be used in sorting.

The method of sorting is the red-black tree method, discussed in most standard texts on algorithms and data structures[12]. It should be noted that this sorting method is 'unstable', i.e. when two rows compare as equal, their relative order in the file is not necessarily preserved. For example, if you sort simply on *_atom_site.label_asym_id* (i.e. the chain id), you cannot rely on the atoms being accessed in residue order, even if they are in that order in the CIF. In such a case, you should sort on both *_atom_site.label_asym_id* and *_atom_site.label_seq_id*.

2.3.1 Searching for values in a CIF.

Once a sort has been set up on a category, it is then possible to perform a search for a loop packet. The essential condition is that all the items whose values are being specified in the search, have been involved in sorting the category (although, within certain limits, it is not necessary to specify values for all the items in the sort list).

The combination of values which is being used as a query for the search (the 'search key') is specifed to the CCIF routine *CCIF_SEARCH_CONTEXT* through three parallel arrays (one for each of real, integer and character values). The members of these arrays

[11] At the present time, this behaviour has not been fully tested or characterised.

[12] See, for example, 'Algorithms in C' by Robert Sedgewick (Addison-Wesley), 1990.

```
       CHARACTER*80 CKEY(5), CKEYBUF*(80*5)
        EQUIVALENCE (CKEY, CKEYBUF)
        INTEGER IKEY(5)
        REAL FKEY(5)
        ...
c ___ Set up sort
        CALL CCIF_SETUP_SORT(...)
        CALL CCIF_ADD_TO_SORT('_atom_site.label_asym_id' ...)
        CALL CCIF_ADD_TO_SORT('_atom_site.label_seq_id' ...)
        ...
        CALL CCIF_DO_SORT(...)

c ___ Set up a context on ATOM_SITE category:
        CALL CCIF_SETUP_CONTEXT(...)

c ___ Now set up search key:

c ___ Chain id is 'A'
        CKEY(1) = 'A'

c ___ Residue id is 250:
        CKEY(2) = ' '
        IKEY(2) = 250

c ___ Now find an atom which satisfies the key:
        CALL CCIF_SEARCH_CONTEXT(INDEX, NCNTXT, 2,
      &  IKEY, FKEY, CKEYBUF, 80, STATUS)
        ...
```

Figure 3: Searching for a residue in a structure.

correspond, in order, to the names in the sort list. So, for example, to search for an atom with a chain id of `A` and a residue number `250`, the key is set up as in figure 3.

After the call to *`CCIF_SEARCH_CONTEXT`*, the context `NCNTXT` will point to a loop packet which contains the values specified in the key, if one exists. If there is no such packet, the context will point to the next packet in the sort order (the value of `STATUS` can be checked to determine if the packet was actually found or not). Data values from this packet can then be manipulated with calls to the *`CCIF_GET_*`* and *`CCIF_PUT_*`* routines, specifying a status of `keep_context`.

To specify a null or unknown value for any data type, set the corresponding member of `CKEY` to `'.'` or `'?'`. These values are checked for immediately on entry to *`CCIF_SEARCH_CONTEXT`* — this is why it is necessary to set `CKEY(2)` to some other value in figure 3.

A search key need not specify values for all the items in the sort list. The condition is that where a sort list has n items, the first m must be specified, where $1 \leq m \leq n$. If a partial key is searched for in this way, on a successful search, the packet which is found is not necessarily the first packet which satisfies the key (neither 'first' in sorted order, nor in the sense of first in the file). This is a consequence of the instability of the sort algorithm referred to above. If it is necessary to find all the packets which satisfy a partial key of m items, a null (`'.'`) value should be given for the items $m+1$ to n. In this case, an exact match may not be found, and the returned value of `STATUS` should not be relied on. The application should check suitable item(s) in the loop packet — see the example program `cifsearch.f` for an example of how this can be done.

2.4 Error reporting and diagnostics

CCIF has a uniform error and diagnostic output mechanism. Every message output by the library has a symbolic name of the form ⟨*facility*⟩_⟨*message*⟩, where ⟨*facility*⟩ identifies the component of the CCIF library that produced the message, and ⟨*message*⟩ identifies the message within that facility. For example, the warning:

```
>>>> CCIF signal ANTLR_EMPTYLOOP (severity: Warning) <<<<
        (Raised in rule "data")
Syntax error at line 16:        Loops with no data not allowed!
```

is identified by the symbolic name `ANTLR_EMPTYLOOP`. (`ANTLR` refers to the parser generator from PCCTS that was used to generate the relevant part of the CCIF code: see part V.) The symbolic name of a particular message is not always output (sometimes just the message appears), but the complete list of messages is in the file `condition.list` in the CCIF source distribution. Each message also has a default severity associated with it (one of: 'Success', 'Informational', 'Warning', 'Error', 'Severe'). Occasionally, the default severity of a particular message is changed to reflect the circumstances under which it is output, but this is rare.

By default, all messages are output, but this is not always appropriate. For example, if a mmCIF contains many data items that are not in the dictionary that is being used, a large number of `CCIF_NOITEMIP` warnings will be generated. Under certain circumstances this may be expected, and the warnings are just a nuiscance. CCIF allows control of the output of messages on a per-message basis, and also on the basis of severity. This can be set both in the environment, or by application programs through the function *`CCIF_SET_MSG`* (see section 8.2).

Each message has one of three output states associated with it: -1 (default), 0 (do not output message) and 1 (output message). Each severity has one of these three states associated with it as well. When the procedure to output a dignostic message is called internally, these states are checked, and the output of the message suppressed if appropriate. The results of this check are given in table 1. The basic idea is that the status of the severity is checked first, but can be overridden for an individual message.

Severity status	Message status	Print message?
-1	-1	Yes
-1	0	No
-1	1	Yes
0	-1	No
0	0	No
0	1	Yes
1	-1	Yes
1	0	No
1	1	Yes

Table 1: Effect of severity and message status codes

This mechanism is bypassed in two cases: firstly, any message that is associated with a condition that causes program execution to stop will always be output. Secondly, the message `CCIF_DICTLOADED` that confirms the loading of the mmCIF dictionary and outputs version information for that dictionary cannot be turned off.

2.4.1 Setting output states using the environment

To set the output state of a message to 1 or 0, set an environment variable (Unix) or logical name (VMS) ⟨*facility*⟩_⟨*message*⟩ to either `ON` or `OFF` respectively (case insensitive). For example:

```
setenv CCIF_NOITEMIP off
```

Any other value (including a null string) will be interpreted as -1 (default).

To set the output state of a severity, use the variable names `CCIF_WARN`, `CCIF_SUCCESS` and `CCIF_INFORM`. `CCIF_ERROR` and `CCIF_SEVERE` are accepted, but have no effect (these severities always cause the program to terminate, and so the message is always output).

Note that the environment-set status can be overridden by the use of the *`CCIF_SET_MSG`* function by the application.

3 Writing programs using CCIF and CCP4

I'll think about writing a step-by-step intro. For now, please look at the example programs cifget.f and cifmodi.f. Note the order in which things are done:

1: Load dictionary with *`CCIF_INIT`*.

2: Open a CIF with *CCIF_LOAD_CIF* and/or *CCIF_NEW_CIF*.

3: Find a block id with *CCIF_BLOCK_BY_NAME* or *CCIF_BLOCK_BY_INDEX*.

4: Initialise context(s) with *CCIF_SETUP_CONTEXT*.

5: Use the data manipulation routines to access individual data values.

Part III

Functions provided by CCIF

This part contains a description of each function provided by the CCIF toolbox. Note that all functions are implemented as C functions of type 'void', so they must be invoked from Fortran programs with the `CALL` statement. Information is passed back to the calling program *via* the parameter list. Each description contains the following sections:

- Usage. This shows how to invoke the function from within a Fortran77 program, and gives the type and status of each parameter. The type of the parameter is given as the form its declaration would take in the function definition, if it was written in Fortran. This is followed by the parameter's status — the possible values are:

  (i): Parameter is input only, and it is usually safe to use a constant expression in the parameter list [13].

  (o): Parameter is output only. It must be the name of a variable of appropriate type, and on return from the function will contain the indicated value. Its value on entry to the function is irrelevant.

  (i/o): Parameter is both input and output. It must be the name of a variable of appropriate type, and before calling the function it should be set to an appropriate value. The function can be expected to set this variable to a different value, as indicated in the description of this parameter and the synopsis.

  Where a parameter is of type `CHARACTER*(*)`, and its status is '(o)' or '(i/o)', it is strongly recommended that it is declared in the calling application as

  `CHARACTER*80` ⟨*variable*⟩

  There is no restriction in mmCIF on the length of data names, or the length of STAR quoted or non-quoted text strings, other than the limit of 80 characters for a single line. (The core CIF restriction of 32 characters for a dataname is *not* enforced in mmCIF.) If the variable supplied is too short to hold the value returned, it will be truncated.

- Synopsis. A description of what the function does, and how various combinations of parameters control its operation.

- Comments. Additional comments, hints on how to use the function, and assorted verbiage (possibly highly opinionated!).

[13] Note for 'C' hackers. The interlanguage calling conventions for all the systems which CCIF supports involve passing parameters either by reference or by descriptor (which is a structure containing a pointer to the value of the parameter). Parameters which are marked as '(i)' are declared in the 'C' function definitions as '`const` ⟨*type*⟩ `* const` ⟨*parameter*⟩', so with an ANSI C compiler, a compile-time check is made that no attempt is made to write to them.

N.B. EXTERN declarations of these functions may be required by convention, or by the compiler options you use to compile your programs — this has not been included in these descriptions. It is normal practice in CCP4 programs to use EXTERN for all identifiers from other modules.

4 File handling functions

This section describes all the CCIF functions which deal with file handling. In CCIF applications, all files are referred to by their logical names only.

4.1 CCIF_INIT

Usage:

```
CALL CCIF_INIT(NAME)
```

CHARACTER*(*) NAME (i): Logical name of dictionary file

Synopsis: Opens the symbol table representation of the CIF dictionary, and makes its contents available to other CCIF routines. The dictionary file will usually be the file mmcifdic.lib. Performs other necessary initialisations for the use of CCIF

Comments: Only one CIF dictionary can be in use at any one time. If you need information from several, you will need to combine all the dictionaries together in one file (more than one data block is permitted), and ensure that each data name defined is unique to the whole dictionary, not just its datablock. This may seem at first sight to be a violation of the STAR scoping rules, but a CIF has no way of identifying which dictionary the definition of a data item comes from, where more than one is available.

It is not an error to call *CCIF_INIT* more than once in an application: the routine remembers if it has been called before, and silently returns without doing anything.

4.2 CCIF_LOAD_CIF

Usage:

```
CALL CCIF_LOAD_CIF(NAME, BLOCKS, IFAIL)
```

CHARACTER*(*) NAME (i): Logical name of CIF.

`INTEGER BLOCKS` (o): On return, set to the number of data blocks in the CIF, -1 if the file could not be opened, or -2 if the function has already been called for that logical name.

`INTEGER IFAIL` (i/o): On input, set to 0 for stop on failure, 1 to allow execution to continue if the file is not found. [This is like the `IFAIL` parameter of CCP4's `CCPDPN` function.] On return, set to 0 for success, -1 if the file could not be opened, -2 if the function has already been called for `NAME`, or to a value greater than 0 if a parsing error was encountered when opening the mmCIF.

Synopsis: This function opens the file pointed to by `NAME`, and performs all lexical and syntactical analysis of its contents, according to the STAR syntax rules. It constructs an in-memory representation of the contents of the CIF, and locates each data name in the file that is defined in the dictionary, issuing warnings for those which are not in the dictionary.

Comments: Only use this function on a file which you know to be a CIF. Its use requires the CIF dictionary to have been previously loaded with *`CCIF_INIT`*. Any attempt to alter the contents of a CIF which has been read with this function will result in a fatal error (c.f. page 10)

4.3 CCIF_NEW_CIF

Usage:

```
CALL CCIF_NEW_CIF(OLDNAME, NEWNAME, BLOCKS)
```

`CHARACTER*(*) OLDNAME` (i): Logical name of input CIF

`CHARACTER*(*) NEWNAME` (i): Logical name of output CIF

`INTEGER BLOCKS` (i/o): On input, like the parameter `BLOCKS` for *`CCIF_LOAD_CIF`* above, with respect to `OLDNAME`. On output, number of blocks in `NEWNAME`

Synopsis: Opens the file pointed to by `NEWNAME`, and initialises it with the contents of `OLDNAME`. If `OLDNAME` consists of one or more spaces, the structures for a new CIF are set up, but the CIF is considered to be empty.

Comments: The contents of the file pointed to by `NEWNAME` can be altered by the CCIF functions described in the following sections. The block id's of the data blocks in `NEWNAME` will be different from those in `OLDNAME`.

4.4 CCIF_PRINT_CIF

Usage:

```
CALL CCIF_PRINT_CIF(NAME)
```

`CHARACTER*(*) NAME` (i): Logical name of CIF

Synopsis: Causes the contents of the CIF `NAME` to be output to the file to which it points.

4.5 CCIF_CLOSE_CIF

Usage:

```
CALL CCIF_CLOSE_CIF(NAME, STATUS)
```

`CHARACTER*(*) NAME` (i): Logical name of CIF

`INTEGER STATUS` (i/o): On input:

0: Write out `NAME`, if it has not already been written out by a previous call to *`CCIF_CLOSE_CIF`* or *`CCIF_PRINT_CIF`*

1: Write out `NAME` regardless.

2: Do not write out `NAME`.

On output:

0: `NAME` not written out

1: `NAME` written out

Synopsis: Closes the logical file entry associated with `NAME`. If no other currently open logical unit refers to the same physical file (see section 2.1), the resources used to store the internal representation of the CIF are freed, and become available to the process once again.

Input values of 0 or 1 for `STATUS` will not cause the file to be written out if it was originally opened read-only (i.e. with a call to *`CCIF_LOAD_CIF`*). A value of 0 will be returned in this case.

An input value of 0 for `STATUS` is sensitive to multiple logical names referring to the same physical file. If the file has been written out already, by using a different logical name, it will not be written out again, even if it has been changed since. Use a value of 1 to force the file to be written out again. (I am open to suggestions as to better ways of handling this.)

Comments: It is a good idea to close a CIF as soon as you have finshed with it. A large CIF can take up a lot of memory.

4.6 CCIF_CHECK_CIF

On its way...

5 Handling blocks and contexts

5.1 CCIF_BLOCK_BY_NAME

Usage:

```
      CALL CCIF_BLOCK_BY_NAME(LOGNAME, BLOCKNAME,
     & BLOCKID, STATUS, DISPOSITION)
```

`CHARACTER*(*) LOGNAME` (i): Logical name of input file

`CHARACTER*(*) BLOCKNAME` (i): Name of data block. Must include the prefixed 'data_' required by the STAR syntax for data block headers.

`INTEGER BLOCKID` (o): Block id (c.f. page 11) corresponding to `BLOCKNAME` in `LOGNAME`.

`INTEGER STATUS` (o): Return status. Possible values are:

- -1: `LOGNAME` has not been opened.
- 0: `BLOCKNAME` not found in file, or does not conform to STAR rules for a data heading.
- 1: `BLOCKNAME` found in file.
- 2: new data block `BLOCKNAME` created in file

`CHARACTER*(*) DISPOSITION` (i): Set to `'NEW'` to create a new data block in the CIF.

Synopsis: If the data block exists in `LOGNAME`, this routine will return its block id. If the value of `DISPOSITION` is `'NEW'` (any mixture of upper and lower case), *and* `BLOCKNAME` does not already exist in `LOGNAME`, *and* `LOGNAME` has been opened for writing, a new, empty, data block with the name `BLOCKNAME` will be created. Other values of `DISPOSITION` are ignored.

Comments: This function is an exception to the rule that an attempt to alter the contents of a CIF which has been opened read-only will cause the application to terminate. There was a good reason for this, but I can't remember what.

5.2 CCIF_BLOCK_BY_INDEX

Usage:

```
      CALL CCIF_BLOCK_BY_INDEX(LOGNAME, IDX, BLOCKNAME,
     & BLOCKID, STATUS)
```

`CHARACTER*(*) LOGNAME` (i): Logical name of CIF

`INTEGER IDX` (i): Sequential number of data block in CIF

`CHARACTER*(*) BLOCKNAME` (o): Name of `IDX`'th data block in `LOGNAME`. Will include the 'data_' prefix as it appears in the CIF.

`INTEGER BLOCKID` (o): Block id of `IDX`'th data block in `LOGNAME`.

`INTEGER STATUS` (o): Return status. Possible values are:

- -1: File `LOGNAME` not open.
- 0: Fewer than `IDX` data blocks in `LOGNAME`, or `IDX` out of range (i.e. < 1).
- 1: Block name and id successfully returned.

Synopsis: Returns the block id and data block header of the `IDX`'th data block in `LOGNAME`. Blocks are numbered from 1 within the CIF.

Comments: It is not possible to create a new data block with this function.

This routine is provided so that CCIF applications can make use of the data blocks without needing to know their names beforehand, or so that keywords such as `ALLBLOCKS` can be made available. It is not a good idea to allow users to specify things like 'the third data block', since the order of data blocks within a CIF is totally arbitrary. User input which specifies particular data block(s), should do so by block name, unless all blocks are being specified.

5.3 CCIF_SETUP_CONTEXT

Usage:

```
      CALL CCIF_SETUP_CONTEXT(NAME, CATNAME, BLOCKID,
     & CONTEXT, STATUS, DISPOSITION)
```

`CHARACTER*(*) NAME` (i): Name of CIF category, or any item which is a member of the category (including aliases defined in the dictionary).

`CHARACTER*(*) CATNAME` (o): True name of category, as declared in the dictionary.

`INTEGER BLOCKID` (i): Block id of data block which context will be used to access.

`INTEGER CONTEXT` (o): Number of context which has been assigned to the category/data block.

`INTEGER STATUS` (o): Information on existence or otherwise of data category in `BLOCKID`. Possible values are:

- -1: Problems with category in CIF prevented context from being set up (e.g. because the category contains items that are not defined in the dictionary).
- 0: Category is not present in `BLOCKID`. A context may have been assigned to this category/data block nevertheless — see the Synopsis.
- 1: Category found in `BLOCKID` in the form of looped data.
- 2: Category found in `BLOCKID` in the form of data items (i.e. STAR data_name data_value pairs).

`CHARACTER*(*) DISPOSITION` (i): Has no effect if CIF was opened for input, i.e. by using *`CCIF_LOAD_CIF`*. Otherwise, two possible values:

- `'RO'`: (Any mixture of upper and lower case) No changes are allowed to be made to the CIF by the use of this context. CCIF will terminate the program with an error if an attempt is made to do this. This allows an application to read back values from a CIF which has been changed, without accidentally altering it again. The value `'READONLY'` is acceptable as a synonym.
- `'LOOP'`: (Any mixture of upper and lower case) If the category of data has key items defined in the dictionary, the sequence of data items will be converted to a STAR loop structure with one row. If the category of data is initially absent from the CIF, a loop structure will be created as data is written to the category..

Synopsis: This is the function used to initialise a context, before using it to read or write data values. If the category is not present in the data block specified, and the CIF has been opened read-only, no context is assigned. A value of -1 is returned in the `CONTEXT` parameter, in this case.

If the CIF has been opened to allow writing, a context is assigned even if the category is not present. The category is actually created by writing values using the *`CCIF_PUT_*`* routines, as described in the next section. This will be done as a sequence of data items, unless `DISPOSITION` was `'LOOP'`, in which case a STAR loop structure will be created.

Comments: The `DISPOSITION` value `'RO'`/`'READONLY'` is provided as a development aid, for checking application program logic. It is not intended that user options should be provided which select this option.

It is the application program's responsibility to retain the value of `CONTEXT` which is being used for a particular purpose.

To create the first packet of a new loop structure, use `append_row` for `STATUS`, in the first call to a *`CCIF_PUT_*`* function. To add further data items to the new packet, use `keep_context` (see table 2).

5.4 CCIF_RELEASE_CONTEXT

Usage:

```
CALL CCIF_RELEASE_CONTEXT(CONTEXT)
```

`INTEGER CONTEXT` (i): Number of context to be freed.

Synopsis: Frees the data structures associated with context `CONTEXT`, allowing the context to be reassigned to another category/data block.

Comments: This is not the only way a context can be freed. See `advance_context` in table 2.

5.5 CCIF_CONTEXT_STATUS

Usage:

```
 CALL CCIF_CONTEXT_STATUS(CONTEXT, BLOCK, CATNAME,
& NCOLS, SINDEX, STATUS)
```

`INTEGER CONTEXT` (i): Number of context.

`INTEGER BLOCK` (o): Block id on which context has been set up.

`CHARACTER*(*) CATNAME` (o): Name of category on which context has been set up.

`INTEGER NCOLS` (o): Number of columns, if the category is a loop; 0 otherwise.

`INTEGER SINDEX` (o): Index of associated sort (see section 7). This will be -1 if the category has not been associated with a sort order.

`INTEGER STATUS` (o): Status of context. Possible values are:

- -1: `CONTEXT` out of range, or not set up.
- 0: Unknown status (indicates that something is wrong).
- 1: Context has been opened read-only
- 2: Context has been opened read-write

Synopsis: Self-explanatory, I hope.... Returns information about a particular context, specified by number

6 Accessing data values

The operation of these routines are controlled with the value of the `STATUS` parameter. The application must set the value to an appropriate value chosen from table 2. The returned value (table 3) provides information on the status of the value. For the *CCIF_GET_** routines, a default value from the dictionary, where available, will always be returned in preference to an unknown value.

Not all the values given in the table are appropriate to every situation. For example, `single_dict` will never be returned for a value which occurs in looped data. Integer dividing by 2 (or a right shift by 1) will allow simpler checking between the various possiblilities, where it is known whether the data is looped or not.

The *CCIF_PUT_** routines do a 'get' on the value internally, before replacing it with the application-supplied value. The returned value of `STATUS` reflects the state of the data just *before* the new value is written out (in this case, `no_value` or `loop_unknown` will be returned, even when there is a default value in the dictionary).

value	mnemonic	description
1	keep_context	Keep context pointing at the same loop packet
2	advance_context	Advance context to next loop packet before accessing value. If previous packet was the last one, or category is not present as a loop, the context will be freed.
3	append_row	Add a packet of unknowns to a loop structure, and point context at that before accessing value. Only valid for the *CCIF_PUT_** routines.
4	advance_and_hold	As `advance_context`, but in the case of looped data, will not free the context on the last row. After a *CCIF_GET_** function returns `end_of_context`, the next *CCIF_PUT_** function should be called with `append_row`.

Table 2: Table of input status values to control the data access routines.

value	mnemonic	description
0	no_value	Data item context: value not present, or specified as '?'
1	loop_unknown	As `no_value`, but in a loop structure
2	single_value	Value returned from a STAR data item
3	loop_value	Value returned from loop packet
4	single_dict	As `no_value`. Returned value is default taken from dictionary
5	loop_dict	As `loop_unknown`. Returned value is default taken from dictionary
6	single_null	Value specified as '.' in a STAR data item.
7	loop_null	Value specified as '.' in loop packet.
8	end_of_context	Ran off end of loop, or item is not in a loop structure (implies input status was `advance_context` or `advance_and_hold`). Returned value is undefined.
9	parenthesis_esd	Special value for *CCIF_GET_REAL_ESD*. Standard deviation was retrieved from parentheses appended to a float data item type. Only applicable where specified by the dictionary
10	invalid_token	Special value for *CCIF_PUT_CHAR*. Could not create a valid STAR token from input provided, so output '.' with warning comments instead. [14]

Table 3: Table of status values returned by the data access routines.

BTYPE	Fortran type	Input function	Output function
0	not determinable	–	–
1	CHARACTER*(*) or CHARACTER*(*)(*)	*CCIF_GET_CHAR* or *CCIF_GET_TEXT*	*CCIF_PUT_CHAR* or *CCIF_PUT_TEXT*
2	INTEGER	*CCIF_GET_INT*	*CCIF_PUT_INT*
3	REAL	*CCIF_GET_REAL*	*CCIF_PUT_REAL*
4	REAL	*CCIF_GET_REAL_ESD*	*CCIF_PUT_REAL_ESD*

Table 4: Values of BTYPE returned by *CCIF_ITEM_TYPE*.

6.1 CCIF_ITEM_TYPE

Usage:

```
      CALL CCIF_ITEM_TYPE(ITEM, CODE, PCODE, BTYPE,
     & SLINE)
```

CHARACTER*(*) ITEM (i): Name of data item

CHARACTER*(*) CODE (o): *_item_type.code* for ITEM

CHARACTER*(*) PCODE (o): *_item_type_list.primitive_code* applicable to ITEM

INTEGER BTYPE (o): Basic type of item. See table 4 for the values and their interpretations.

INTEGER SLINE (o): For character/text data, a value of 0 indicates that the value of ITEM may be multi-line text, and a value of 1 that the value can only appear on a single line.

Synopsis: Retrieves information from the ITEM_TYPE_LIST dictionary category relating to ITEM.

Comments: Use this routine if you are unsure which of the *CCIF_GET_** or *CCIF_PUT_** functions to use to handle ITEM. BTYPE and SLINE together define which functions to use — see table 4.

[14]This will soon be changed so that a check is made against the construct from the dictionary as well.

6.2 Items common to the CCIF_GET_* functions.

`CHARACTER*(*) ITEM` (i): Name of data item whose value is to be retrieved.

`CHARACTER*(*) CVALUE` (o): Text of value of `ITEM` as it is present in the CIF. Returned for diagnostic purposes

`INTEGER CONTEXT` (i/o): Context from which the value of `ITEM` is to be retrieved

`INTEGER STATUS` (i/o): Status parameter

Synopsis: Retrieves value of `ITEM` from `CONTEXT`.

Comments: `CONTEXT` is '(i/o)', since it will be set to -1 if the context is freed.

6.3 CCIF_GET_INT

Usage:

```
      CALL CCIF_GET_INT(ITEM, CVALUE, VALUE,
     & CONTEXT, STATUS)
```

`INTEGER VALUE` (o): Value of `ITEM` as an integer.

6.4 CCIF_GET_REAL

Usage:

```
      CALL CCIF_GET_REAL(ITEM, CVALUE, VALUE,
     & CONTEXT, STATUS)
```

`REAL VALUE` (o): Value of `ITEM` as a real.

6.5 CCIF_GET_REAL_ESD

Usage:

```
  CALL CCIF_GET_REAL_ESD(ITEM, CVALUE, VALUE,
& CESD, ESD, CONTEXT, STATUS, ESDSTATUS)
```

`REAL VALUE` (o): Value of `ITEM` as a real.

`CHARACTER*(*) CESD` (o): As for `CVALUE`, but for associated esd, where it was present in the CIF as a separate data value.

`REAL ESD` (o): Value of esd associated with `ITEM`.

`INTEGER ESDSTATUS` (o): Status value for esd (see table 3).

Synopsis: This function retrieves a value and its associated esd, auto-detecting whether it is present as a parenthesised value, or a separate data item or data value.

Comments: At the present time, the mmCIF dictionary does not allow parenthesised esd's for values which are defined to be of type `int`. (This does not forbid a token like `1234(5)`, where *`_item.type_code`* is defined to be `float`, of course.)

6.6 CCIF_GET_CHAR

Usage:

```
  CALL CCIF_GET_CHAR(ITEM, CVALUE, VALUE,
& LENVAL, CONTEXT, STATUS)
```

`CHARACTER*(*) VALUE` (o): Value of `ITEM` as a character. If the token in the CIF was single quoted or double quoted, the quotes and any leading spaces or tabs will have been removed. (Since Fortran character variables are always right-padded, trailing space is irrelevant.)

`INTEGER LENVAL` (o): Length of data returned in `VALUE`, not counting quotes or leading/trailing spaces.

6.7 CCIF_GET_TEXT

Usage:

```
 CALL CCIF_GET_TEXT(ITEM, ITMPOS, NLINE, BUF,
& NLBUF, CONTEXT, STATUS)
```

`INTEGER ITMPOS` (i/o): Placemarker within text.

`INTEGER NLINE` (o): Number of lines of text actually retrieved (but see synopsis).

`CHARACTER(*) BUF` (o): Buffer to receive text.

`INTEGER NLBUF` (i): Number of text lines `BUF` can hold. `LEN(BUF)` must be an integral multiple of `NLBUF` — a value of 80 for this multiple is strongly suggested. If it is less, lines within the retrieved text will be truncated, or the data may be otherwise garbled.

Synopsis: Where data is present in the CIF as multi-line text, a maximum of `NLBUF` lines is retrieved, starting `ITMPOS` characters from the start. Each line will be padded with spaces to the end of line, the line length being `LEN(BUF)/NLBUF`. The returned value of `ITMPOS` will be the position of the end of the last line returned in `BUF`, except where no more text remains, when the value of `ITMPOS` will be zero. This allows the text to be retrieved in chunks, with `ITMPOS` being retained by the application program between calls to this function. The bounding semi-colons are not stripped from the data returned.

Where data is present in the CIF as a single-, double-, or non-quoted text string, `BUF` will contain the value with quoting removed, and `NLINE` will contain the length of the data (i.e. equivalent to `VALUE` and `LENVAL` respectively for *`CCIF_GET_CHAR`*). In this case, the whole of the data will be returned from each call, irrespective of the input value of `ITMPOS`, which will be retured with a value of -1. Since this is fundementally a STAR, rather than a CIF issue, this case also coveres to the case where the value is unknown/unspecified or null, in addition to setting `STATUS` appropriately.

Comments: Handling text of arbitrary length with Fortran77 is a pain — please don't blame me! No matter how large you dimension `BUF`, the text which you are trying to retrieve may not all fit in. Part of the problem is that you don't know in advance whether the data is actually present as multi-line text, or as a quoted or non-quoted text string, so this routine has to cope with both cases.

The best way to cope with the buffer is to equivalence it to a character array, for example:

```
      CHARACTER*80 BUF*800, TEXT(10)
      EQUIVALENCE(TEXT,BUF)
         ....
      ITMPOS = 0
10    CALL CCIF_GET_TEXT('_citation.abstract', ITMPOS, NLINE,
     & BUF, 10, CONTEXT, STATUS)

      IF (ITMPOS .LT. 0) THEN
C___DO SOMETHING WITH SINGLE-LINE TEXT TOKEN HERE.
         ...
         GOTO 30
      ENDIF


C____DO SOMETHING WITH LINES OF TEXT HERE
     ...

      IF (ITMPOS .GT. 0) GOTO 10

C____FINISHED DEALING WITH THIS TEXT
30    CONTINUE
```

This will give you up to 10 lines of text on each call, which you can then handle line-wise by using `TEXT`. There is no obligation to retrieve the whole of a text string — you can give up before you reach the end.

6.8 Items common to the CCIF_PUT_* functions.

`CHARACTER*(*) ITEM` (i): Name of data item whose value is to be output.

`INTEGER CONTEXT` (i/o): Context to which the value of `ITEM` is to be output

`INTEGER STATUS` (i/o): Status parameter

Synopsis: Replaces current value of `ITEM` in `CONTEXT` with `VALUE`.

Comments: `CONTEXT` is '(i/o)', since it will be set to -1 if the context is freed.

6.9 CCIF_PUT_REAL

Usage:

```
CALL CCIF_PUT_REAL(ITEM, VALUE, CONTEXT, STATUS)
```

`REAL VALUE` (i): Value of `ITEM` to output.

6.10 CCIF_PUT_REAL_ESD

Usage:

```
 CALL CCIF_PUT_REAL_ESD(ITEM, VALUE, ESD, CONTEXT,
& STATUS, DISPOSITION)
```

`REAL VALUE` (i): Value of `ITEM` to output.

`REAL ESD` (i): Value of esd of `ITEM` to output.

`CHARACTER*(*) DISPOSITION` (i): If the first 5 characters of this parameter are `'PAREN'` (any mixture of upper and lower case) *and* the dictionary definition of `ITEM` allows a parenthesised esd (i.e. *`_item_type_conditions.code`* is `esd`), the value of esd will be output appended to the value in parentheses. Otherwise, the esd will be output as a separate value, to the data item for which *`_item_related.function_code`* is `associated_esd`.

Comments: My opinion is that using parenthesised esd's in CIF's is A Bad Thing. This is because the relationship between the value proper, and its esd has to be explained in some external definition, which then has to be hard-coded into any software which accesses the data. Using a separate data item for the esd, on the other hand, allows the relationship between the value and its esd to be defined in the dictionary, and requires no special extra code to handle it.[15]

6.11 CCIF_PUT_INT

Usage:

```
CALL CCIF_PUT_INT(ITEM, VALUE, CONTEXT, STATUS)
```

`INTEGER VALUE` (i): Value of `ITEM` to output.

[15] This does not apply to situations where the data is presented through a viewer, e.g. a special CIF editor, or a program which prepares tables for publication. In this case, parenthesised esd's are A Good Thing, since they are concise and easily understood.

6.12 CCIF_PUT_CHAR

Usage:

```
CALL CCIF_PUT_CHAR(ITEM, VALUE, CONTEXT, STATUS)
```

`CHARACTER*(*) VALUE` (i): Value of `ITEM` to output.

Comments: Use this function to output a non-, single-, or double-quoted text string (including '?' or '.'), even when the item type allows multi-line text. To output multi-line text for those items which allow it, use *`CCIF_PUT_TEXT`*.

The input value can be quoted if desired, but it is not necessary. If the value is not already a valid STAR token, this function will attempt to make it into one, by trying single-quoting and then double-quoting it, if the length is 78 characters or less. If these both fail, *and* the item allows multi-line text, the function will then try creating a multi-line token as follows:

```
;
⟨VALUE⟩
;
```

If all these fail, the function will issue a warining, and output a null value, with STAR comments preceding the token flagging the error and giving the offending text.

6.13 CCIF_PUT_TEXT

Usage:

```
 CALL CCIF_PUT_TEXT(ITEM, NLINES, BUF, NLBUF,
& CONTEXT, STATUS, DISPOSITION)
```

`INTEGER NLINES` (i): Number of lines in `BUF` to be output

`CHARACTER*(*) BUF` (i): Buffer holding text to be output

`INTEGER NLBUF` (i): Actual number of lines in `BUF`

`CHARACTER*(*) DISPOSITION` (i): If the value of this parameter is `'NEW'`, any previous text will be replaced with the contents of `BUF`. Any other value of this parameter will cause the contents of `BUF` to be appended to existing text.

Synopsis: The first `NLINES` lines of `BUF` will be output to the value of `ITEM` on `CONTEXT`, according to the value of `DISPOSITION`. For the relationship between `BUF` and `NLBUF`, see the description of *`CCIF_GET_TEXT`* (section 6.7).

Comments: It is unnecessary to put bounding semi-colons around the supplied text — after each call to this function, a semi-colon will have been appended, so the text stored internally is always valid. Putting a semi-colon at the end of the text, requires you to be absolutely sure that `NLINES` is consistent with the position of the semi-colon, otherwise you will lose text. This happens because the function terminates the input text at the first line it finds which begins with a semi-colon, even if less than `NLINES` lines have been output (this is to ensure compliance with the STAR rules). Text after this semi-colon will be lost.

Having a semi-colon as the very first character of `BUF` when `DISPOSITION` is `'NEW'` is acceptable. All things considered, though, it is safest not to put in the bounding semi-colons yourself at all, and let them be inserted automatically.

6.14 Special considerations for handling esd's.

The use of *`CCIF_GET_REAL`*/*`CCIF_PUT_REAL`* is acceptable for data which might have an esd associated with it. Careful use of *`CCIF_GET_REAL_ESD`* followed by *`CCIF_PUT_REAL`* will avoid expanding a CIF with unnecessarily specified esd's.

If you are going to use the information you read from a context, to produce output, it is best to use *`CCIF_GET_REAL_ESD`* to get the value, and check `ESDSTATUS`. If `ESDSTATUS` indicates that a value for the esd came from the input file, use *`CCIF_PUT_REAL_ESD`* to write it out again. If `ESDSTATUS` indicates that the esd was null, unknown, or the default value from the dictionary, use *`CCIF_PUT_REAL`* to output the value.

This is an important consideration where some operation is carried out on the values. For example, in applying a transformation to an atomic coordinate, the rotational part of the transformation must be applied to the esd's as well. See the treatment of coordinates in `cifsymop` for an example of how this can be done in practice.

6.15 CCIF_COPY_ROW

Usage:

```
CALL CCIF_COPY_ROW(CXIN, CXOUT, STATUS)
```

`INTEGER CXIN` (i): Context from which row is to be copied.

`INTEGER CXOUT` (i): Context to which row is top be copied.

`INTEGER STATUS` (i/o): Status value to be applied to `CXOUT` (see tables 2 and 3).

Synopsis: Copies all the values from the loop packet pointed to by `CXIN` to `CXOUT`. Obviously, these two contexts must refer to the same category, or corresponding categories in different datablocks or files. `CXIN` and `CXOUT` can refer to the same category within a datablock (and even point to the same row, in which case nothing is done), but `CXIN` cannot be equal to `CXOUT`.

Comments: I hope to provide similar functions to copy larger groups of data.

At the moment, nothing is done to check that the combination of key items in the row output to the loop structure is uniquely defined.

7 Sorting and searching

7.1 CCIF_SETUP_SORT

Usage:

```
CALL CCIF_SETUP_SORT(NAME, BLOCK, INDEX, STATUS)
```

`CHARACTER*(*) NAME` (i): Name of category (or any item within the category) to be sorted.

`INTEGER BLOCK` (i): Block id which contains `NAME`

`INTEGER INDEX` (o): Index of sort.

`INTEGER STATUS` (o): 1 for success; 0 for failure.

Synopsis: Initialises a data structure to hold information on sort. The index of the sort is returned in the `INDEX` parameter, and should be maintained by the calling application.

Comments: The maximum number of sorts which can be active at once is controlled by the preprocessor symbol `NSORT_TREES`.

7.2 CCIF_ADD_TO_SORT

Usage:

```
CALL CCIF_ADD_TO_SORT(NAME, INDEX, STATUS)
```

`CHARACTER*(*) NAME` (i): Name of item to add to sort list.

`INTEGER INDEX` (i): Index of sort list (from a previous call to *CCIF_SETUP_SORT*).

`INTEGER STATUS` (o): 1 for success; 0 for failure.

Synopsis: Adds the item `NAME` to the end of the sort list referred to by `INDEX`. To sort a category on several items, call this routine repeatedly — each call adds the data name to the end of the sort list.

Comments: No sorting is allowed on item names which may have text (i.e. multi-line) values.

If this routine is called more than once for the same `INDEX` with the same `NAME`, the name is moved to the end of the list.

7.3 CCIF_DO_SORT

Usage:

```
CALL CCIF_DO_SORT(INDEX, STATUS)
```

`INTEGER INDEX` (i): Index of list to be applied.

`INTEGER STATUS` (o): 1 for success; 0 for failure.

Synopsis: Sorts loop packets in ascending order. When comparing two loop packets, the values are compared in the order in which they occur in the sort list (as set up by *CCIF_ADD_TO_SORT*).

Comments: If no data names have been supplied to the sort list, a list is constructed from the category's key item(s). If there is more than one key item in the category, they are applied in the order in which they are defined in the CIF dictionary, which was used to prepare the file loaded by *CCIF_INIT* (see section 4.1).

Further items may be added to a sort list, and the sort order refined, by additional calls to *CCIF_ADD_TO_SORT* and *CCIF_DO_SORT*.

7.4 CCIF_RELEASE_SORT

Usage:

```
CALL CCIF_RELEASE_SORT(INDEX, STATUS)
```

`INTEGER INDEX` (i/o): Index of sort list to be released. Returned as -1 if release was successful.

`INTEGER STATUS` (i/o): Controls behaviour, when contexts are still linked to the sort list. Values on input:

0: Will not release sort if it is still linked to any contexts. The number of linked contexts will be returned in `STATUS`.

1: Unlink contexts from this sort, without changing the packets to which they refer. The number of contexts affected will be returned in `STATUS` as a negative number.

2: Unlink contexts from this sort, resetting them to refer to the first loop packet. The number of contexts affected will be returned in `STATUS` as a negative number.

Synopsis: Frees the data structures used by the sort `INDEX`, and makes them available for a new sort.

7.5 CCIF_SORT_CONTEXT

Usage:

```
CALL CCIF_SORT_CONTEXT(INDEX, NCNTXT, STATUS)
```

`INTEGER INDEX` (i): Index of sort list.

`INTEGER NCNTXT` (i): Number of context.

`INTEGER STATUS` (i/o): On input: 0 to keep context referring to current packet; 1 to make context refer to first packet by sort order. On output: 1 for success; 0 for failure.

Synopsis: Connects a previously set up context to a sort order. When the context is advanced, it is to the next row in the sort order, rather than to the following packet in the loop.

Comments:

7.6 CCIF_SEARCH_CONTEXT

Usage:

```
 CALL CCIF_SORT_CONTEXT(INDEX, NCNTXT, NVALS,
& IKEY, FKEY, CKEYBUF, CKEYLEN, STATUS)
```

`INTEGER INDEX` (i): Index of sort list.

`INTEGER NCNTXT` (i): Number of context.

`INTEGER NVALS` (i): Number of values in key.

`INTEGER IKEY(*)` (i): Array of at least size `NVALS`, containing values of type integer for searching.

`REAL FKEY(*)` (i): Array of at least size `NVALS`, containing the values of type real for searching.

`CHARACTER CKEYBUF*(*)` (i): Character buffer of at least `NVALS*CKEYLEN`, containing the values of type character for searching. Each value is in a portion of the buffer of size `CKEYLEN`.

`INTEGER CKEYLEN` (i): Number of characters of `CKEYBUF` occupied by each value.

`INTEGER STATUS` (o): 1 for key found; 0 for key not found.

Synopsis: Searches through the sort given by `INDEX` for a row matching the given key (see section 2.3.1 for details of how to construct a key). The parameters `NVALS`, `IKEY`, `FKEY`, `CKEYBUF` and `CKEYLEN` collectively define a key. The context given by `NCNTXT` (which must have been previously set up on the same category in the same data block as the sort) is left refering to the 'first'[16] row which matches the key, if the returned value of `STATUS` is 1. If the value is 0, the context will point to the next highest row in sort order.

Comments: If the context is linked to a sort already, it is still possible to use a different sort on the same category for searching. Subsequent calls to *`CCIF_GET_*`* or *`CCIF_PUT_*`* routines which advance the context, will use the sort order to which the context has been linked, to determine which row to advance to.

[16]This may not be the first matching row in the file. See the comments about the sorting algorithm in section 2.3.

8 Miscellaneous functions

8.1 CCIF_OUTPUT_FMT

Usage:

```
CALL CCIF_OUTPUT_FMT(ITEM, FLAGS, WIDTH, PREC, FMT, STATUS)
```

`CHARACTER*(*) ITEM` (i): Name of data item.

`CHARACTER*(*) FLAGS` (i): Flag characters, that modify format conversion. Must be zero or more of `-, +, _, 0` (`_` is converted to a space).

`INTEGER WIDTH` (i): Width part of format specification.

`INTEGER PREC` (i): Precision part of format specification.

`CHARACTER*1 FMT` (i): Format conversion character for output item. Must be one of `i, d, f, e, E, g, G`.

`INTEGER STATUS` (o): Returned status for operation. Possible values are:

- -2: `FMT` is invalid for some reason. Default format string will be used for this item, namely `"%d"`, `"%i"` and `"%s"` for real, integer and character data respectively.
- -1: `ITEM` not in dictionary. Call ignored with warning.
- 0: `FMT` applied to `ITEM`.

Synopsis: Most of the *CCIF_PUT_** functions use the 'C' `sprintf(3)` function to write the supplied data values into holding buffers, from which they are output to the CIF proper by *CCIF_PRINT_CIF*. This fuction can be used to specify the format string which will be used for this purpose, for any data item. The format string is generated by assembling the flags, width, precision and format conversion character, into a full `sprintf` format specifier, e.g. `'%-7.4f'`. A negative value may be specified for the width or precision, if they are to be omitted. The precision is ignored for integer or character data types. The format specification character is ignored for character data types – `s` is always applied.

This mechanism cannot be used to influence the behaviour of *CCIF_PUT_TEXT*. The application program is responsible for the layout of multi-line text.

If this routine is called for a category, rather than an item, a value of `'n'` for `FMT` will force every loop packet in that category to start on a new line on output. Any other string will prevent the loop packets from starting a new line (except to comply with the 80 characters to a line rule).

When this function has not been called for a particular category, and the category has been substantially copied from an input file, CCIF will attempt to reproduce something like the input format. A new category which has been built up using CCIF routines will normally start each loop packet on a new line[17].

Comments: It is not intended to provide sophisticated formatting control in CCIF — this is really in the domain of CIF viewers/visualisers, rather than software which manipulates the data. Nevertheless, it can sometimes be useful to maintain the elements of some familiar style, or restrict the number of significant digits output, to avoid rounding errors becoming visible.

For an item which can append an esd in parentheses, `FMT` should not contain any items for the output of the esd, just for the value proper. Where an esd is to be output as a separate data item, this function should be called for the associated esd item, as well as for the main item.

If this method is used to develop a standard style for more than just a few data items, it would be best to put ⟨*item name*⟩ ⟨*format specifier*⟩ items into a file, and read from this at run time, rather then hard-coding everything into the application program.

8.2 CCIF_SET_MSG

Usage:

```
CALL CCIF_SET_MSG(NAME, STATUS, OLDSTATUS)
```

`CHARACTER(*) NAME` (i): name of message or severity

`INTEGER STATUS` (i): Output status to assign to message or severity

`INTEGER OLDSTATUS` (o): Previous status. A value of -2 indicates that `NAME` was not recognised.

Synopsis: Sets the output status of a message or severity to `STATUS`, returning the previous status in `OLDSTATUS`. See section 2.4 for a discussion of error and diagnostic messages in CCIF. The values for `NAME` are the same as those of the environment variables or logical names discussed in section 2.4.1.

[17] In general, CCIF does not attempt to be too sophisticated about formatting CIF output. If you want pretty-looking CIF data, write a CIF viewer!

Comments: Using this function overrides the corresponding environment settings, i.e. if you set a message to always output, the user of your program will not be able to turn it off.

8.3 CCIF_SET_LINE_LIMIT

```
CALL CCIF_SET_LINE_LIMIT(NAME, LINELEN)
```

`CHARACTER*(*) NAME` (i): Logical name of CIF.

`INTEGER LINELEN` (i): Maximum length of line for `NAME`

Synopsis: This function changes the maximum line length allowed for a mmCIF on output, effectively controlling the wrapping. The practical limit on the specified length is the C preprocessor symbol `ZZLEXBUFSIZE`, by default 2000.

Part IV

Things CCIF does not do (yet).

- When converting from data items to a loop, CCIF checks that the category has key items defined, but does not check that any of those items actually exist in the data.
- Check that where a mmCIF item of type 'code' or 'ucode' has a restricted set of possible values given by _item_enumeration.value, that the values in the CIF are acceptable.
- Update the _audit.update_record automatically.
- Checking that all the mandatory items of data in a category are present (sub categories make this more complicated than it seems at first sight).
- Deal properly with category groups and sub categories
- Some 'higher level' CIF checking, for example checking that there are no duplicated n-tuples of key items in a category, i.e. that each loop packet can be uniquely identified.

Part V

Acknowlegements

- This work was funded mainly by CCP4, but I continued some work on it at the EBI.
- The STAR scanner and parser were generated with PCCTS (the Purdue Compiler Construction Tool Set) v 1.33. Information is available at `http://www.antlr.org`. I have also had valuable correspondence with the author Terence Parr, and the maintainer of the 1.33 release, Tom Moog.
- The implementation of the red-black tree sorting/searching algorithm is 'librb' by James Plank (from volume28 of comp.sources.misc).