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

Fig. 5.4.7.1. Example application accessing aliased data names.

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

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

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

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

The execution of this code results in the following printout.

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

5.4.8. Implementation of the tools

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

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

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

5.4.9. How to read CIF data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4.10. How to write a CIF

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