The GAN Generalized Driver

Robert ROY* and Alain HÉBERT
Institut de génie nucléaire
École Polytechnique de Montréal
*e-mail: roy@meca.polymtl.ca

March 2000

Abstract

This new version of the GAN generalized driver can be used to call a sequential series of modules sharing a common calling convention. It provides a template to build FORTRAN scientific applications by linking independent modules, each one performing an elementary task. Data can only be exchanged between modules using files or linked lists, an abstract data type that features a hierarchical structure. A linked list is an auto-descriptive information structure kept in core memory and referenced using a single pointer. A complete set of standard utility modules is provided to perform operations on the files and linked lists. Any application can be customized by adding application-dependent and specific utility modules. Any application based on the GAN generalized driver will share a common user interface and will behave in a standard manner.

Contents

List of Figures

1 Introduction

1.1 The GAN generalized driver: an overview

A scientific application can be build around the GAN generalized driver by linking it with application-dependent modules.[1, 2] Such a scientific application will share the following specifications:

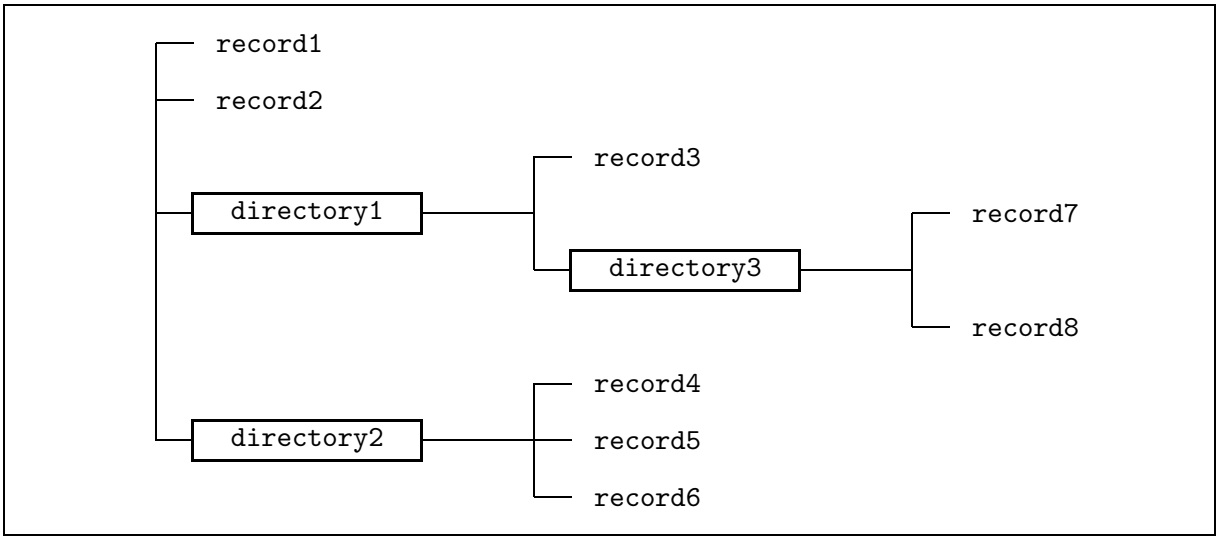

Figure 1: An example of a linked list.

1. The advanced kernel can handle a custom data type, implemented as a linked list. Each data type mapped to a hierarchical structure, is dynamically allocated using the computer's memory management algorithm and is accessed with a pointer. Linked lists are the *only* memory-resident data type used to transfer information between modules (common blocks and `INCLUDE` statements will no longer be used). An example of a linked list is shown in Fig. 1. However, interface files can be used to transfer information between modules in cases where we want to reduce the memory resource requirements. XSM files are available to store any hierarchical structure similar to the example shown in Fig. 1.

2. Building a scientific application requires the definition of the linked lists and interface files and the programming of application-dependent modules to manage these linked lists or files.

3. The application-dependent and utility modules can be programmed in one of the following languages:

   - straight FORTRAN-77
   - FORTRAN-77 with MIL-STD 1753 extensions (Cray-style pointers, structures and records)
   - FORTRAN-90
   - C
   - Ada

   This new version of the advanced kernel was written in straight FORTRAN-77, but has also been translated into FORTRAN-90, now that suitable compilers are available.

4. A kernel for the driver was written to support the linked lists and to read macro-language instructions. The modules are callable from this driver, but the possibility of having "embedded modules", i.e. modules called directly from a subroutine written in any of these four languages has also been introduced.

5. Utility modules are available to backup the linked lists on an XSM file and to permit code restart.

With this new user interface, the input to a module named `MOD:` with two embedded modules `EMB1:` and `EMB2:` will always be of the form:

(list of output linked lists) := `MOD:` *(list of input linked lists)* `::` *(data input for MOD:)*
`:::` `EMB1:` *(data input for EMB1:)* `;`
`:::` `EMB2:` *(data input for EMB2:)* `;`
`;`

This approach will make it possible to perform complex simulations and the debugging of modular parts of our codes will be easier. The modular calculation strategy will also ensure that subsequent developments can be easily implemented in a fully-integrated computation environment. The concern that students can learn and be involved in developing codes in a more

user-friendly environment is also an important benefit. The kernel of the generalized driver is closely connected to the CLE-2000 compiler that will now be described.

1.2 CLE-2000 and the generalized driver

The CLE-2000 control language allows loops, conditional testing and macro-processor capabilities to be included in the generalized driver input deck. A reversed polish notation (RPN) calculator named `EVALUATE` is also provided. An example of conditional testing is shown in the following example involving two modules:

```
INTEGER INDEX ;
MODULE MOD1:  MOD2:  ;
.
.
.
EVALUATE INDEX := 0 ;
REPEAT
  EVALUATE INDEX := INDEX 1 + ;
  IF INDEX 3 > THEN
    (output list) := MOD1: (input list) :: (data input for MOD1:) ;
  ELSE
    (output list) := MOD2: (input list) :: (data input for MOD2:) ;
  ENDIF ;
UNTIL INDEX 7 >= ;
```

This type of programming provides the user with much more flexibility than the conventional approaches. It is possible to build new applications without re-compilation, simply by changing the order of the module calls and by making modifications to the conditional logic. It is very simple to develop a user-defined function even if this possibility is not programmed into any module.

The CLE-2000 control language brings the following capabilities to any code:

- `INTEGER`, `REAL`, `STRING`, `DOUBLE` and `LOGICAL` declarations to contain control language and macro-processor variables. The language provides no direct data type conversion.
- macro processor variables. For example, it is possible to define a variable `VAR1` as equal to a real number and to use `<<VAR1>>` in place of this real number later on.
- reversed polish notation calculator. A calculator is called upon each time the statement `EVALUATE` is used. For example, the statement

  ```
  EVALUATE VAR1 := 4.0 6.0 + ;
  ```

  would assign the result 10.0 to the variable `VAR1`. Logical operations are fully supported.
- a simple printer. For example, the variable `VAR1` can be printed using the command

  ```
  ECHO VAR1 ;
  ```

- three types of control loops. The available control loops are:
  - `IF` *(logical expression)* `THEN` *(user instructions)* `ELSE` *(user instructions)* `ENDIF ;`
  - `REPEAT` *(user instructions)* `UNTIL` *(logical expression)* `;`
  - `WHILE` *(logical expression)* `DO` *(user instructions)* `ENDWHILE ;`

Note that the `EVALUATE` and `ECHO` statements are *not* modules of the generalized driver.

1.2.1 CLE-2000 instructions

Users can code CLE-2000 instructions only in the 72 first character positions of any line of the source file. All other entries (columns 73 and up) should no longer be used. Comments can also be included in the source file; they usually begin with **!** (exclamation). Except for the comment lines, blanks are significant; they are used in order to separate variables, operations, keywords,etc. All CLE-2000 instructions must end with a semicolon **;** normally followed by a carriage return. The simple fact that the semicolon may not end an instruction specific to CLE-2000 is not important to the compiler. After each semicolon, the compiler will identify

which instructions have a CLE-2000 meaning, and other instructions are sent to the kernel. The CLE-2000 compiler is responsible in producing a compiled file, where conditional logic is manageable and where computations results in data of the correct type anywhere in the source file. Links with the applications, as input/output access, and executions of the compiled file is no longer its role.

1.2.2 Values and variables

Any value accepted by the CLE-2000 compiler has one of the following five types: logical, integer, real, double or string. When analyzing a possible value, CLE-2000 determines its type according to:

Integer any sequence of decimal numbers (unsigned integer);

Integer a sign + or − followed by any sequence of decimal numbers (signed integer);

Real a sequence of decimal numbers with one decimal point;

Real a sequence of decimal numbers, preceded or not by a sign and with or without a decimal point, with an E followed by a signed or unsigned integer;

Double sequence of decimal numbers, preceded or not by a sign and with or without a decimal point, with an D followed by a signed or unsigned integer ;

String a sequence of characters enclosed between two quotation marks (character ”) or two apostrophes (character ’) (or any sequence with no blank that was not identified as a numeric value by the previous items) ;

Logical a type mostly restricted to CLE-2000 statements (with internal values $True_L or $False_L).

There are value limits for these different types. The following restrictions are imposed on the content of a string:

- the length of any string is restricted to 72 characters;

- a string must not contain an exclamation mark (character **!** reserved for comments);
- a string must not contain double IO symbols (this excludes characters << or >> whose functions will be described later);
- in CLE-2000 statements, strings must be enclosed between quotation marks (character **"**);
- outside CLE-2000, strings are character sequences that were not identified as numeric (with no blank) or character sequences between apostrophes (character **'**).

These limits are imposed by the language in which the CLE-2000 compiler was written. For the actual 2.1 version, Fortran was used and the usual limits for integers, real and double precision data apply. Strings are restricted to 72 characters.

Variable names are restricted to 12 alphanumeric values. Once declared, these variable identifiers are uniquely associated with a memory location in a direct access file. Every name must begin with a letter or an underscore, the other values may be letters, digits or underscores. A particular type of variable is a pre-defined constant (or parametric) which begins with a $ sign (each application can have its own set of parametric constants). Once declared, these variable identifiers are uniquely associated with a memory location in the direct access (object) file. At any step in the CLE-2000 source file, the user may store a value in the variable using an `EVALUATE` statement; moreover, the developer of an application can also store a value in the variable using the syntax `>>VAR1<<`. He may also recall the value of a variable inside any evaluation/calculation step.

1.2.3 Interface between CLE-2000 and the kernel

The interface between CLE-2000 and this kernel (or any application module) is done using $<< \cdot >>$ or $>> \cdot <<$ syntax. To access the content of a variable *VAR_IN* , the access instruction is:

Structure **(inputmode)**

$<<VAR\_IN>>$

When developing a new module that will fully use the CLE-2000 syntax, the developer can also use the inverse access in order to put a value in a CLE-2000 variable. To put a value into a variable VAR_OUT , the inverse access instruction is:

Structure **(outputmode)**

$>> VAR\_OUT <<$

Note that this inverse access operation is done after checking if the type correctly matched the one given by the user.[4]

2 The kernel syntax (user's guide)

Using the generalized driver automatically gives the user access to a set of standard utility modules. These modules perform general purpose actions on the linked lists and files. Some of these modules are declaration modules, others only give information on data, and finally some modules act on linked lists and/or files (either by changing or copying some data).

2.1 Syntactic rules for input data specifications

This user's guide was written using the following conventions:

- the parameters surrounded by single square brackets '[ ]' denote an optional input;
- the parameters surrounded by double square brackets '[[ ]]' denote an optional input which may be repeated as many times as desired;
- the parameters in braces separated by vertical bars '{ | | }' denote a choice of input where (one and *only* one is mandatory);
- the parameters in **bold face** and in brackets '( )' denote an input structure;
- the parameters in italics and in brackets with an index '(*data*(i), i=1,n)' denote a set of n inputs;
- the words using the typewriter font are character constants `KEY WORDS` used as key words;
- the words in italics are user-defined variables, they should be lower case and are of type integer (starting with *i* to *n*) and floating point (starting with *a* to *h* or *o* to *z*) or of type character in uppercase *CHARACTER*.

In order to simplify the presentation of generalized-driver *statements*, the semicolon will be simply withdrawn from the description of each sentence.

2.2 Declaration modules

In the GAN generalized driver, there exists eight different types available. Object of these types must be declared only once in a source file; they are static in the sense that, once defined, their values will stay available anywhere in the source file. The declarations can be given anywhere in the source file. It is important to note that an object must be declared before its first appearance in an executable statement. There are two forms of declarations: the simple declaration of the type and the declaration followed by some specifications on the objects.

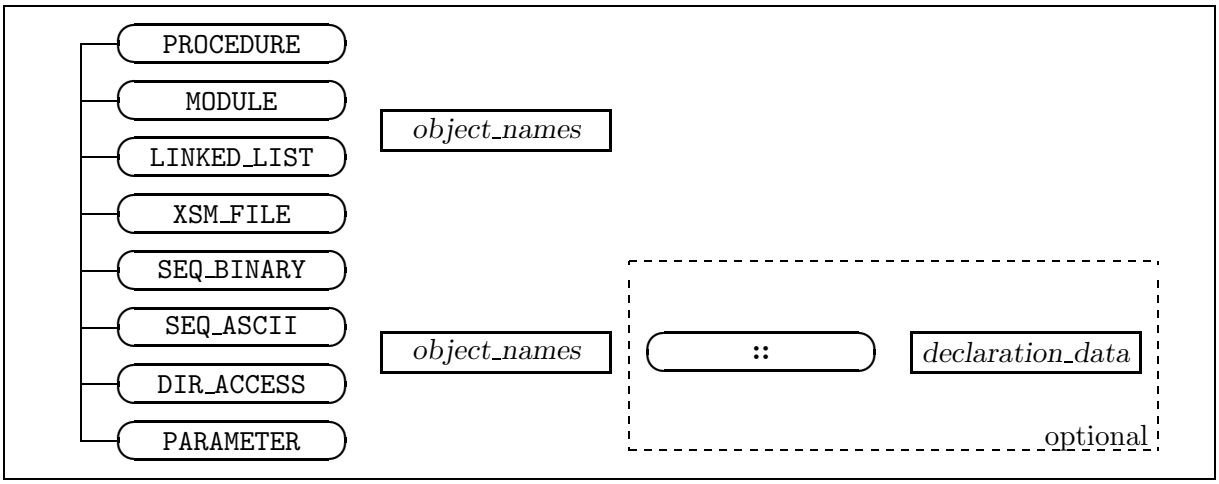

Figure 2: The declaration modules.

For the current version, the full declaration form including some *declaration_data* is available for files declared by `SEQ_BINARY` , `SEQ_ASCII` and `DIR_ACCESS` and in procedure parameters using `PARAMETER` . The structure of all these types will now be described. Input to a scientific application build around this GAN generalized driver must always follow the following specifications.

2.2.1 The `PROCEDURE` declaration

The kernel of the generalized driver offers the possibility of user-defined procedures. These procedures give the user the possibility to "program" an application using the full capabilities of the generalized driver in a calling procedure. A procedure is a set of input data specifications contained in a distinct ASCII file and called from another input data file. If the command `PROCEDURE␣file` has been issued by the user, three files with different extensions are involved

in the compilation process:

- `file.c2m` is the name of the ASCII source file to be compiled;
- `file.o2m` is the name of the object file resulting from the compilation;
- `file.l2m` is the name of the output file used for comments and errors.

The name of the procedure must correspond with the name of the ASCII *file without extension.* If there exists no file named `file.o2m` in the execution directory, the compilation will proceed from the ASCII file `file.c2m`:

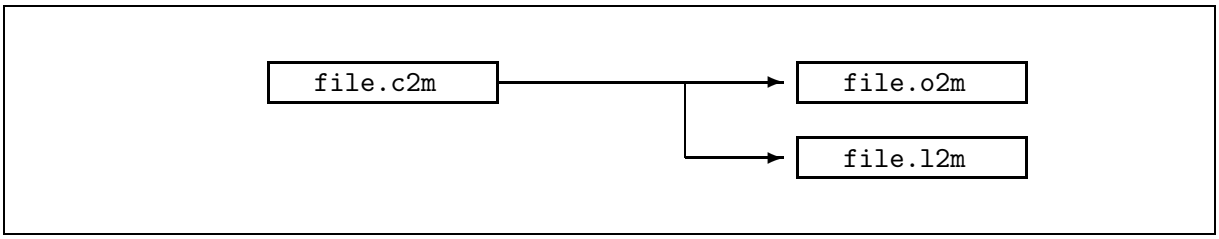

Figure 3: Compilation process.

The input to this `PROCEDURE` declaration takes the form of the structure **(procedure)** :

Structure **(procedure)**

`PROCEDURE` [[*NAME_PR*]]

NAME_PR `character*12` name of a procedure available in the execution directory. Note that procedures must be declared before their first use.

Here is an example of a simple procedure declaration statement:

```
PROCEDURE FACT ;
```

2.2.2 The `MODULE` declaration

The `MODULE` declaration gives the names of modules that will be used later in the input source. Every module in the input source must be declared before it is first used. The input to this declaration module takes the form of the structure **(module)** :

Structure **(module)**

`MODULE` [[*NAME_MD*]]

NAME_MD `character*12` name of a module available in the application. Note that the embedded modules do not need to be declared.

Here is an example that gives the list of all executable modules available in this kernel:

```
MODULE ADD:  BACKUP: DELETE:   END:  EQU:  FINDO:  FREE:  GREP:
       IOX:  MPX:    RECOVER:  STAT: UTL:  ;
```

2.2.3 The `LINKED_LIST` declaration

The `LINKED_LIST` declaration gives the names of linked lists that will be used later in the input source. Every linked list in the input source must be declared before it is first used. The input to this declaration module takes the form of the structure **(linkedlist)** :

Structure **(linkedlist)**

`LINKED_LIST` [[*NAME_LL*]]

NAME_LL `character*12` name of a linked list.

Here is an example of a simple linked list declaration statement:

```
LINKED_LIST LL ;
```

2.2.4 The `XSM_FILE` declaration

The `XSM_FILE` declaration gives the names of XSM files that will be used later in the input source. Every XSM file in the input source must be declared before it is first used. The input to this declaration module takes the form of the structure **(xsmfile)** :

Structure **(xsmfile)**

`XSM_FILE` [[*NAME_XF*]]

NAME_XF `character*12` name of a XSM file.

Here is an example of a simple XSM file declaration statement:

```
XSM_FILE XF ;
```

2.2.5 The `SEQ_BINARY` declaration

The `SEQ_BINARY` declaration gives the names of sequential binary files that will be used later in the input source. Every sequential binary file in the input source must be declared before it is first used. The input to this declaration module takes the form of the structure **(seqbinary)** :

Structure **(seqbinary)**

```
SEQ_BINARY [[ NAME_SQ ]]
[ :: [ EDIT impr ]
FILE [[ FILE_SQ ]]
]
```

NAME_SB `character*12` dummy name of a sequential binary file. Note that the dummy name is the true name if the full data form is not used.

`EDIT` key word used to set the *impr* variable to 0 (no printing in the module) or 1 (some printing is done in the module). The default value is *impr* = 0.

`FILE` key word used before actual file names given in *FILE_SQ* . There must be the same number of file names than the number of sequential binary dummy names. The actual file names are limited to `character*72` data.

There are thus two forms for declaring sequential binary files: in the short form, the dummy name is limited to 12 characters and is equal to the actual true file name; in the long form, the dummy name is used to access a file which actual file name can have up to 72 characters. Here is an example of a long declaration:

```
SEQ_BINARY  MYLIB              OUT               ::
      FILE '/home/roy/libroy' '/home/roy/outbin'  ;
```

2.2.6 The `SEQ_ASCII` declaration

The `SEQ_ASCII` declaration gives the names of sequential ASCII files that will be used later in the input source. Every sequential ASCII file in the input source must be declared before it is first used. The input to this declaration module takes the form of the structure **(seqascii)** :

Structure **(seqascii)**

```
SEQ_ASCII [[ NAME_SA ]]
[ :: [ EDIT impr ]
FILE [[ FILE_SA ]]
]
```

NAME_SA `character*12` dummy name of a sequential ASCII file. Note that the dummy name is the true name if the full data form is not used.

`EDIT` key word used to set the *impr* variable to 0 (no printing in the module) or 1 (some printing is done in the module). The default value is *impr* = 0.

`FILE` key word used before actual file names given in *FILE_SA* . There must be the same number of file names than the number of sequential ASCII dummy names. The actual file names are limited to `character*72` data.

Here is an example of a long-form declaration for two ASCII files:

```
SEQ_ASCII   IN                  OUT               ::
      FILE '/home/roy/input' '/home/roy/output'   ;
```

2.2.7 The `DIR_ACCESS` declaration

The `DIR_ACCESS` declaration gives the names of direct access files that will be used later in the input source. Every direct access file in the input source must be declared before it is first used and the record length of each file must be given. The input to this declaration module takes the form of the structure **(diraccess)** :

Structure **(diraccess)**

```
DIR_ACCESS [[ NAME_DA ]]
:: [ EDIT impr ] [ FILE ]
RECL [[ [ FILE_DA ] recl ]]
```

NAME_DA `character*12` dummy name of a direct access file. Note that the dummy name is the true name if the full data form is not used.

`EDIT` key word used to set the *impr* variable to 0 (no printing in the module) or 1 (some printing is done in the module). The default value is *impr* = 0.

`FILE` key word used before actual file names given in *FILE_DA* . There must be the same number of file names than the number of direct access dummy names. The actual file names are limited to `character*72` data.

`RECL` mandatory key word used to set the *recl* integers. There must be the same number of *recl* than the number of names. The record lengths are given in 32-bit words (that is usually in multiple of 4 bytes).

Here is an example of a long-form declaration for a 1024-byte direct access file:

```
DIR_ACCESS WIMSLIB ::
           FILE RECL '/home/roy/lib/WIMSLIB' 256 ;
```

2.2.8 The `PARAMETER` declaration

The `PARAMETER` declaration gives the dummy names of parameters (linked lists or files) that were passed by a calling program into a procedure and that will be used later in the input procedure. The input to this declaration module takes the form of the structure **(parameter)** :

Structure **(parameter)**

```
PARAMETER [[ NAMPRM ]]
[ :: [ EDIT impr ]
[ ::: LINKED_LIST [[ NAME_LL ]] ; ]
[ ::: XSM_FILE [[ NAME_XF ]] ; ]
[ ::: SEQ_BINARY [[ NAME_SB ]] ; ]
[ ::: SEQ_ASCII [[ NAME_SA ]] ; ]
[ ::: DIR_ACCESS [[ NAME_DA ]] ; ]
]
```

NAMPRM `character*12` dummy name of a parameter. Note that the parameter inherits its declaration type from the calling program in the short form.

EDIT	key word used to set the *impr* variable to 0 (no printing in the module) or 1 (some printing is done in the module). The default value is *impr* = 0.
LINKED_LIST	embedded module called to verify if actual arguments *NAME_LL* passed by the calling program are linked lists.
XSM_FILE	embedded module called to verify if actual arguments *NAME_XF* passed by the calling program are XSM files.
SEQ_BINARY	embedded module called to verify if actual arguments *NAME_SB* passed by the calling program are sequential binary files.
SEQ_ASCII	embedded module called to verify if actual arguments *NAME_SA* passed by the calling program are sequential ASCII files.
DIR_ACCESS	embedded module called to verify if actual arguments *NAME_SA* passed by the calling program are direct access files.

In the long form, the parameter must satisfy the proper type when this input file procedure is executed. Note that some parameters may inherit their types, at the same time as other parameters are forced to satisfy some types. Here is an example where we used 6 linked lists and 2 sequential binary files:

```
PARAMETER       FLUX6 EDIT6 LIBRARY CANDU6F TRACK6F TRACK6S
                                            TRCAN6F TRCAN6S ::
::: LINKED_LIST FLUX6 EDIT6 LIBRARY CANDU6F TRACK6F TRACK6S ;
::: SEQ_BINARY                              TRCAN6F TRCAN6S ;
                                                            ;
```

It is important to note that there is a semicolon at the end of each embedded module, so that the final sentence ends with two semicolons for the long form of declaration.

2.3 Standard modules

A complete set of standard utility modules is provided to perform usual operations on linked lists or files. Applications will then be customized by adding application-dependent and specific modules without caring about how to transmit data between linked lists or files. In Fig. 4, we depict the list of the standard modules. Note that in some cases (modules `EQU:` and `IOX:`) it is not necessary to use the name of the module.

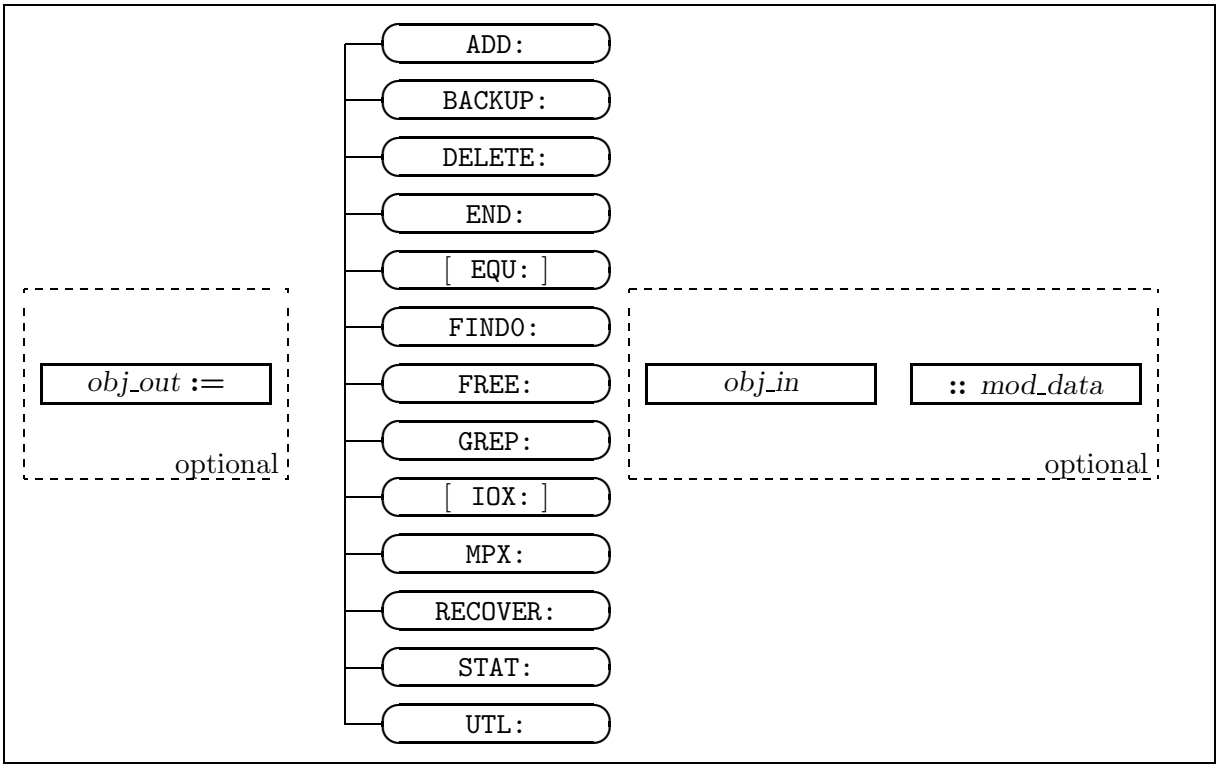

Figure 4: The standard modules.

Some standard modules have only input objects (linked lists or files), others have output objects. There are standard modules where users can input some data; others do not allow any data input. We will now describe shortly each of these modules.

2.3.1 The `EQU:` module

This module is used to copy a linked list, an XSM file, a sequential binary file or a sequential ASCII file. The calling specifications for structure **(equality)** are:

Structure **(equality)**

NAME1 [[*NAME2*]] `:=` [`EQU:`] *NAME3* [`::` `EDIT` *impx*]

NAME1 `character*12` name of a linked list, an XSM file, a sequential binary file or a sequential ASCII file.

NAME2 `character*12` name of a linked list, an XSM file, a sequential binary file or a sequential ASCII file. Optional data.

NAME3 `character*12` name of a linked list, an XSM file, a sequential binary file or a sequential ASCII file.

`EDIT` key word used to set the *impx* variable to 0 (no printing in the module) or 1 (some printing is done in the module). The default value is *impx* = 1.

The name of this module does not have to be formally given. The kernel is able to detect whenever there is a sentence with linked lists or file names but no module and thus to associate this operation to equality. The LHS is (are) created. If both the RHS and LHS are linked lists or XSM files, a single copy is performed. A linked list can be created from an XSM file or an XSM file can be created from a linked list. If the LHS is a sequential file and the RHS is a linked list or an XSM file, an export is performed. The export format is either binary or ASCII, depending on the type of the LHS file. If the LHS is a linked list or an XSM file and the RHS is a sequential file, an import is performed. The case where both the LHS and the RHS are sequential files is not supported.

2.3.2 The `UTL:` module

The UTL: module is used to perform utility actions on a *single* linked list or a *single* XSM file. *NAME1* is either created, modified or read, depending on where it appears on each side of `UTL:` . The calling specifications for structure **(util)** are:

Structure **(util)**

[*NAME* :=] UTL: [*NAME*] ::
[[DIR]] [[STEP { UP *NOMDIR* | DOWN }]]
[[IMPR *BLOCK* { *ileni* | * }]]
[[CREA *BLOCK ilenc* = { (*valc* (i),i=1,*ilenc*) | (*ivalc* (i),i=1,*ilenc*) | (*hvalc* (i),i=1,*ilenc*)
| (*dvalc* (i),i=1,*ilenc*) }]]
[[MULT *BLOCK flott*]]
[[{ COPY | STAT { REL | ABS } | ADD } *NOMREF NOMALT*]]
[DUMP]

NAME1 character*12 name of a linked list or an XSM file that will be treated by the utility module.

DIR key word used to print the active directory content.

STEP key word used to move into the XSM hierarchy.

UP key word used to move up towards a sub-directory of the active directory.

NOMDIR name of the sub-directory to which we wish to head.

DOWN key word, allows descent towards the sub-directory containing the active directory.

IMPR key word, used to evaluate the number of non zero elements in a block located on *NAME1* and to print it in whole or in part. The only accessible blocks are those found on the active directory of *NAME1* .

MULT key word, allows each element of a block or sub-directory in the active directory to be MULTiplied by a real constant. If *BLOCK* is a sub-directory, only floating point information contained in it is multiplied.

CREA key word used to CREAte a block of information on *NAME1* . This block will be created in the active directory.

BLOCK name of the block or sub-directory chosen.

ileni maximum number of elements that the user wishes to print. A value of *ileni* =0 is permitted.

`*` key word, indicates that all the elements of a block will be printed. In a realistic case, the number of elements contained in a block may be rather large; this option must therefore be used with caution.

ilenc number of elements which the newly created block will contain.

`=` key word, indicates that the input values will follow.

valc real vector containing the information to be written to *NAME1* .

ivalc integer vector containing the information to be written to *NAME1* .

hvalc character*4 array containing the information to be written to *NAME1* .

dvalc double precision array containing the information to be written to *NAME1* .

flott constant by which a block or sub-directory present on *NAME1* will be multiplied.

`COPY` key word used to copy an existing block (or sub-directory) administered by XSM onto a new block (or sub-directory) existing or not. If it does not exist, it will be created automatically.

`ADD` key word used to add the contents of two blocks or two sub-directories on *NAME1* . If *NOMREF* and *NOMALT* are sub-directories, only the floating point information contained in them is added. The result is written in *NOMALT* .

`STAT` key word used to compare the contents of two blocks on *NAME1* .

`REL` the relative differences are printed. Neither of the two blocks is modified.

`ABS` the absolute differences are printed. Neither of the two blocks is modified.

NOMREF name of the reference block.

NOMALT name of the block which may possibly be modified during the `ADD` and `COPY` operations.

`DUMP` Dump the active directory of *NAME1* and its sub-directories to the printer. For example, a linked list (or XSM file) named `OBJ10` can be dumped to the printer using the following command:

```
UTL: OBJ10 :: DUMP ;
```

2.3.3 The `DELETE:` module

This module is used to delete one or many linked lists, XSM files, sequential or direct access files. The calling specifications for structure **(delete)** are:

Structure **(delete)**

NAME1 [[*NAME2*]] `:=` `DELETE:` *NAME1* [[*NAME2*]]

NAME1 `character*12` name of a linked list, an XSM file, a sequential binary file, a sequential ASCII file or a direct access file.

NAME2 `character*12` name of a linked list, an XSM file, a sequential binary file, a sequential ASCII file or a direct access file. Optional data.

The names of the linked lists and files should be written on both the LHS and the RHS. Only linked lists and files that do not break any dependency rule can be deleted.

2.3.4 The `BACKUP:` module

This module is used to copy one or many linked lists (or XSM files) together with its (their) dependencies to a backup media. The backup media can be a single linked list, XSM file, sequential binary file or sequential ASCII file. The calling specifications for structure **(backup)** are:

Structure **(backup)**

NAME1 `:=` `BACKUP:` [*NAME1*] [[*NAME2*]] [`::` `EDIT` *impx*]

NAME1 `character*12` name of a linked list, an XSM file, a sequential binary file or a sequential ASCII file. *NAME1* is used as a backup media.

NAME2 `character*12` name of a linked list or an XSM file that is to be backuped.

`EDIT` key word used to set the *impx* variable to 0 (no printing in the module) or 1 (some printing is done in the module). The default value is *impx* = 1.

If *NAME1* appears only on the LHS, it is created. If *NAME1* appears on both the LHS and the RHS, it is updated.

2.3.5 The `RECOVER:` module

This module is used to recover one or many linked lists (or XSM files) previously saved on a backup media. The backup media can be a single linked list, XSM file, sequential binary file or sequential ASCII file. The calling specifications for structure **(recover)** are:

Structure **(recover)**

[[*NAME1*]] `:=` `RECOVER:` *NAME2* [[*NAME1*]] [`::` `EDIT` *impx*]

NAME1 `character*12` name of a linked list or an XSM file that is to be recovered.

NAME2 `character*12` name of a linked list, an XSM file, a sequential binary file or a sequential ASCII file. *NAME2* is used as a backup media.

`EDIT` key word used to set the *impx* variable to 0 (no printing in the module) or 1 (some printing is done in the module). The default value is *impx* = 1.

If *NAME1* appears only on the LHS, it is created. If *NAME1* appears on both the LHS and the RHS, it is replaced by the information located on the backup media.

2.3.6 The `FREE:` module

This module was originally defined and used to free one or many linked lists or files from their dependencies with their daughters. The calling specifications for structure **(free)** are:

Structure **(free)**

[[*NAME1*]] `:=` `FREE:` [[*NAME1*]]

NAME1 `character*12` name of a linked list or a file that we want to free from its dependencies. Each name should appear on both the LHS and the RHS.

In this new version of the generalized driver, the dependencies are no longer support. However, to insure that old source files can run, the `FREE:` module was kept and it now does absolutely nothing.

2.3.7 The `ADD:` module

This module is used to add the floating point information contained in two linked lists or XSM files located on the RHS. The result is stored in *NAME1* . The calling specifications for structure **(addition)** are:

Structure **(addition)**

NAME1 `:=` `ADD:` { *NAME1 NAME2* | *NAME2 NAME3* }

NAME1 `character*12` name of a linked list or an XSM file that will contain the result of the addition.

NAME2 `character*12` name of a linked list or an XSM file that is not modified.

NAME3 `character*12` name of a linked list or an XSM file that is not modified.

2.3.8 The `MPX:` module

This module is used to multiply the floating point information contained in a linked list or an XSM files located on the RHS by a user-defined real number. The result is stored in *NAME1* . The calling specifications for structure **(multiply)** are:

Structure **(multiply)**

NAME1 `:=` `MPX:` { *NAME1* | *NAME2* } `::` *real*

NAME1 `character*12` name of a linked list or an XSM file that will contain the result of the multiplication.

NAME2 `character*12` name of a linked list or an XSM file that is not modified.

real real number used to multiply the linked list or the XSM file.

2.3.9 The `STAT:` module

This module is used to compare the floating point information contained in two linked lists or XSM files named *NAME1* and *NAME2* . The calling specifications for structure **(statistics)** are:

Structure **(statistics)**

```
STAT: NAME1 NAME2
```

NAME1 `character*12` name of a the first linked list or XSM file.

NAME2 `character*12` name of a the second linked list or XSM file.

2.3.10 The `GREP:` module

The GREP: module is used to extract a *single* value from a linked list or XSM file. The calling specifications for structure **(grep)** are:

Structure **(grep)**

```
GREP: NAME ::
[[ STEP { UP NOMDIR | DOWN } ]]
[[ { GETVAL | MAXVAL | MINVAL | INDMAX | INDMIN | MEAN }
BLOCK index1 [ { { index2 | * } [ index3 ] | NVAL { nval | * } } ]
[[ >>VAR_IN<< ]] ]]
```

NAME `character*12` name of a linked list or an XSM file from which an extraction will be performed.

`STEP` key word used to move into the XSM hierarchy.

`UP` key word used to move up towards a sub-directory of the active directory.

NOMDIR name of the sub-directory to which we wish to head.

`DOWN` key word, allows descent towards the sub-directory containing the active directory.

`GETVAL` key word used to get values from an existing block of a linked list or an XSM file. The receiving CLE-2000 variables are assumed to be of the same type as the picked values (all CLE-2000 types are supported).

`MAXVAL` key word used to get the maximum value from a list of values for an existing block of a linked list or an XSM file. The receiving CLE-2000 single variable is assumed to be of the same type as the picked maximum (valid for integer, real and double precision types).

`MINVAL` key word used to get the minimum value from a list of values for an existing block of a linked list or an XSM file. The receiving CLE-2000 single variable is assumed to be of the same type as the picked minimum (valid for integer, real and double precision types).

`INDMAX` key word used to get the index (position inside the block) of the maximum value from a list of values for an existing block of a linked list or an XSM file. The receiving CLE-2000 single variable is assumed of an integer type (valid for integer, real and double precision blocks).

`INDMIN` key word used to get the index (position inside the block) of the minimum value from a list of values for an existing block of a linked list or an XSM file. The receiving CLE-2000 single variable is assumed of an integer type (valid for integer, real and double precision blocks).

`MEAN` key word used to get the mean value from a list of values for an existing block of a linked list or an XSM file. The receiving CLE-2000 single variable is assumed to be of the same type as the computed mean (valid only for real and double precision types).

BLOCK name of the block chosen in which the extraction will be made.

index1 the first element to be extracted is the *index1* -th element of *BLOCK* .

index2 if *index2* is given, the last element to be extracted will be the *index2* -th element of *BLOCK* ; if the extracted value is of character type, the *index1* -th to *index2* -th elements of *BLOCK* will be extracted and concatenated in a single character variable (of maximum length of 72). Default value: 1.

`*` if `*` is given, the last element to be extracted is effectively the last element in *BLOCK* .

index3 if *index3* is given, it will specify the step between values to be extracted

between *index1* and *index2* . Default value: 1.

`NVAL` key word used to specify the number of elements to be extracted from the block

nval this value is the number of elements to be extracted from the block; if the extracted value is of character type, the *index1* -th to *index1* +*nval* -1-th elements of the block are extracted.

VAR_IN `character*12` CLE-2000 variable name in which the extracted value will be placed. It is expected that the number of values extracted and the number (and types) of variables agree with the extraction instruction.

2.3.11 The `FIND0:` module

The `FIND0:` module is used to find the root of a function using the Brent's method. This procedure assumes that the zero is bracketed in an interval given as input using the two first points, and that the function used is continuous in this interval. The calling specifications for structure **(findzero)** are:

Structure **(findzero)**

NAME `:=` `FIND0:` [*NAME*] `::`
{ [`DEBUG`] [`ITMAX` *itmax*] [`TOL` *tol*] `POINT X` *x1* `Y` *y1* `POINT X` *x2* `Y` *y2* | `Y` *y3* }
>>*CONV_L* << >>*NEWZERO_R* <<

NAME `character*12` name of a linked list or an XSM file (type `L_0`) that will contain all information necessary for the zero-finding procedure. If *NAME* appears on both sides, it is updated; otherwise, it is created.

`DEBUG` key word used to edit the content of most variables in the `L_0` object; used only for debugging purposes.

`ITMAX` key word used to specify the maximum number of iterations that will be allowed for the zero-finding procedure. The procedure will abort if the number of iterations goes beyond this maximum value.

itmax the maximum number of iterations. Default value: 100.

`TOL` key word used to specify the tolerance on the zero to be found.

tol tolerance. Default value: 1.E-5.

`POINT` key word used to specify that the next point will be given.

`X` key word used to specify that an abscissa will be given.

`Y` key word used to specify that an ordinate will be given.

x1 the first abscissa value.

y1 the first ordinate value.

x2 the second abscissa value.

y2 the second ordinate value.

y3 in the case we are in an update mode, only a new ordinate value is given.

CONV_L `character*12` name of a CLE-2000 variable. The *CONV_L* parameter is expected to be a CLE-2000 logical variable, that will contain the convergence mode (true if the zero-finding procedure has converged, false otherwise. In creation mode, this value will normally be set to false.

NEWZERO_R `character*12` name of a CLE-2000 variable. The *NEWZERO_R* parameter is expected to be a CLE-2000 real variable, that will contain the next value (to approximate zero) to be used as in the zero-finding procedure. In creation mode, this value will be computed using the two points given as input; when it is modified, the procedure checks if it receives the last given value. It is important to used this particular value for the next function evaluation.

Note that the zero-finding procedure has an initial mode where *NAME* is created. In the initialization process, the two points specifying the interval must be given, and it is expected that $y1 \times y2 < 0$. In the updated mode, there is no need to put back the abscissa of the next point because it is expected to be the last real value that was generated by the procedure. This explains why you will only input `Y` *y3* .

The `L_0` specification is used to store intermediate values needed by the zero-finding procedure `FIND0:` . There are no directories in this object, and it is created and updated only by the `FIND0:` module. To understand the content of the object, it is possible, using the labels

given for every block, to refer to Brent's algorithm as given in Ref. [3]. Here is a description of the content for the basic directory of the linked-list produced by this module:

Directory name: '/ '

begin records description

'SIGNATURE	'	Character*12 signature of the linked list or XSM file. Always equal to 'L_0'.
'X	'	Abscissae for 3 points. Dimension: 3 (values labelled *a*, *b*, *c*).
'Y	'	Ordinates for 3 points. Dimension: 3 (values labelled *f(a)*, *f(b)*, *f(c)*).
'DE	'	Intermediate real values. Dimension: 2 (labelled *d*, *e*).
'PQRS	'	Intermediate real values. Dimension: 4 (labelled *p*, *q*, *r*, *s*).
'ITERV	'	Integer vector containing iteration driving data. Dimension: 4 (values labelled *iter*, *itmax*, *iprt*). *iter* is the iteration number, *itmax* is the maximum number of iterations allowed and *iprt* is set to 1 in debugging mode, 0 otherwise.
'TOL	'	Real vector containing tolerance data. Dimension: 3 (values labelled *tol*, *xm*, *tol1*). *tol* is the tolerance permitted on the zero, *xm*, *tol1* are intermediate values related to tolerance.

end records description

The following example shows how to use this method to find a zero:

```
* CLE-2000 VERS 2.1 * R.ROY, EPM COPYRIGHT 1999 *                          LINE
*                                                                          0001
*--------------------------------------------------------------------------0002
*                                                                          0003
* FIND THE ROOT SOLUTION OF " X= EXP(-X) ".                               0004
*                                                                          0005
*--------------------------------------------------------------------------0006
*                                                                          0007
LINKED_LIST L0 ;                                                          0008
MODULE FIND0: END: ;                                                      0009
REAL Y1 Y2 ROOT YNEW ;                                                    0010
REAL X1 := 0.0 ;                                                          0011
REAL X2 := 1.0 ;                                                          0012
```

```
LOGICAL CONV ;                                                          0013
*                                                                       0014
EVALUATE Y1 := X1 CHS EXP X1 - ;                                        0015
EVALUATE Y2 := X2 CHS EXP X2 - ;                                        0016
L0 := FIND0: :: ITMAX 20 TOL 1.0E-4                                     0017
                POINT X <<X1>> Y <<Y1>>                                 0018
                POINT X <<X2>> Y <<Y2>>                                 0019
                >>CONV<< >>ROOT<< ;                                     0020
REPEAT                                                                  0021
   EVALUATE YNEW := ROOT CHS EXP ROOT - ;                               0022
   L0 := FIND0: L0 :: Y <<YNEW>>                                        0023
                    >>CONV<< >>ROOT<< ;                                 0024
UNTIL CONV ;                                                            0025
ECHO "Zero is =" ROOT ;                                                 0026
QUIT .                                                                  0027

>|Zero is =  5.670696E-01                                             |>0026
 .-----------------------------------------------------------------------.
```

At line number 15 and 16, we evaluate two values of the function $f(x) = e^{-x} - x$ which are such that $f(x_1) \times f(x_2) < 0$. This is sent to the creation step defined defined at lines 17-20. An approximation of the root r is found. A sequence of calculations is now done: at every step, the new value of f is computed using the best root yet available, and this is followed by a new call to the `FIND0:` module to update root values.

2.3.12 The `IOX:` module

The `IOX:` module is used to recover input parameters (macro-processor variables) from the calling statement and to return values to the calling program. This module can only be used in a procedure. The calling specifications for structure **(ioexternal)** are:

Structure **(ioexternal)**

[`IOX:`] :: [[{ >>*VAR_IN*<< | *VAL_OUT* }]]

VAR_IN `character*12` name of a CLE-2000 declared variable. The next value in the input stack of the calling program is copied in the variable; types in the calling program and for this variable are expected to match.

VAL_OUT Any value acceptable in the CLE-2000 syntax can be returned to the calling program. Once more, the type of the recipient is supposed to match with this value.

Note that the module name itself does not have to be given. If a sentence contains data, without any linked list or file name, the kernel will associate the default module `IOX:` . This default value is the best way to use this module without having to declare it.

In order to show the IO transfers between the calling program and procedures, we will take a simple example where we want to compute 8! Suppose that the main input file contains:

```
*
* Calling the recursive "FACT" procedure:
*
*  input  to  "FACT": *n*
* output from "FACT": *n_fact*
*
*    use to compute n!
*
 PROCEDURE FACT      ;
 INTEGER   n := 8    ;
 INTEGER   n_fact    ;
*
* call with IO values.
 FACT ::    <<n>> >>n_fact<< ;
 ECHO "FACT(" n ")="  n_fact ;
 QUIT .
```

Once executed, this input file will generate an output where we will find an echo of the lines. This output should look like this:

```
* CLE-2000 VERS 2.1 * R.ROY, EPM COPYRIGHT 1999 *                          LINE
*                                                                          0001
* Calling the recursive "FACT" procedure:                                  0002
*                                                                          0003
*  input  to  "FACT": *n*                                                  0004
* output from "FACT": *n_fact*                                             0005
*                                                                          0006
*    use to compute n!                                                     0007
*                                                                          0008
 PROCEDURE FACT      ;                                                    0009
 INTEGER   n := 8    ;                                                    0010
 INTEGER   n_fact    ;                                                    0011
*                                                                          0012
* call with IO values.                                                    0013
 FACT ::    <<n>> >>n_fact<< ;                                            0014
 ECHO "FACT(" n ")="  n_fact ;                                            0015
 QUIT .                                                                   0016
```

On line number 13, this main input calls the procedure `FACT`; the first data is a value that will be passed to the procedure, the second value is an output value that will be recovered from the procedure. In this procedure (supposed to be a file called `FACT.c2m`), the access to the IO stack can be done by:

```
 !
 ! Example of a recursive procedure.
 !
 !  input  to  "FACT": *n*
 ! output from "FACT": *n_fact*
 !
 INTEGER n n_fact prev_fact ;
 !
 !   *n* is recovered from the calling program
 :: >>n<< ;
 !
 IF n 1 = THEN
```

```
  EVALUATE n_fact := 1 ;
 ELSE
  PROCEDURE FACT ;
  EVALUATE n := n 1 - ;
  !
  ! Here, "FACT" calls itself
  FACT ::   <<n>>  >>prev_fact<< ;
  EVALUATE n_fact := n 1 + prev_fact * ;
 ENDIF ;
 !
 !   *n_fact* is returned to the calling program
 :: <<n_fact>>  ;
 !
 QUIT .
```

whose output (once compiled) should look as:

```
* CLE-2000 VERS 2.1 * R.ROY, EPM COPYRIGHT 1999 *                          LINE
 !                                                                         0001
 ! Example of a recursive procedure.                                       0002
 !                                                                         0003
 !  input  to  "FACT": *n*                                                 0004
 ! output from "FACT": *n_fact*                                           0005
 !                                                                         0006
 INTEGER n n_fact prev_fact ;                                            0007
 !                                                                         0008
 !   *n* is recovered from the calling program                           0009
 :: >>n<< ;                                                              0010
 !                                                                         0011
 IF n 1 = THEN                                                           0012
   EVALUATE n_fact := 1 ;                                                0013
 ELSE                                                                    0014
   PROCEDURE FACT ;                                                      0015
   EVALUATE n := n 1 - ;                                                 0016
   !                                                                     0017
   ! Here, "FACT" calls itself                                           0018
   FACT ::   <<n>>  >>prev_fact<< ;                                      0019
   EVALUATE n_fact := n 1 + prev_fact * ;                                0020
 ENDIF ;                                                                 0021
 !                                                                         0022
 !   *n_fact* is returned to the calling program                         0023
 :: <<n_fact>>  ;                                                        0024
 !                                                                         0025
 QUIT .                                                                  0026
```

In line number 10, we recover the value sent by the main program. In line number 24, we return the computed value to the main program.

Note that the number (and all types) of arguments must be the same in the calling program and at the end of the procedure. However, the arguments can be recovered (and returned) in batches. Arguments are processed from left to right according to their status (I/O) in the main program. It is a good habit to keep input values, before output values. Be aware that expected output values no longer exist in the main program until they are sent back at the end of the procedure call. This is more or less similar to any other high-level programming language.

2.3.13 The `END:` module

This module is used to delete all the local linked lists, to close all the remaining local files and to return from a procedure or to stop the run. The calling specifications for structure **(end)** are:

Structure **(end)**

```
END:
```

2.4 User's directives

The following user's directives are always followed by an application built around the generalized driver:

- A linked list is resident in core memory if declared as `LINKED_LIST` in the input data or mapped in a direct access file (of XSM type) if declared as `XSM_FILE` in the input data.
- All the information declared as `LINKED_LIST` is destroyed at the end of a run. All other information is located on files which are kept at the end of the run, unless explicitly destroyed by a `DELETE:` command.
- Consider the following example in which the module `MOD1:` is called with the following command:

  ```
  DATA1 DATA2 := MOD1: DATA2 DATA3  ;
  ```

  Linked lists or files on which `DATA2` and `DATA3` exhibit a dependency are also available in read-only mode to the module `MOD1:`.
- In the previous example, `DATA1` is opened in **create** mode because it appears only on the left-hand side (LHS) of the command. `DATA2` is opened in **modification** mode because it appears on both sides of the command. Finally, `DATA3` is opened in **read-only** mode because it appears only on the right-hand side (RHS) of the command.
- This example produces a first dependency of `DATA1` on `DATA2` and `DATA3` and a second dependency of `DATA2` on `DATA3`. In older versions of the driver, the modification of `DATA3` was prohibited as long as `DATA1` and `DATA2` were not deleted. Any modification to `DATA2` was also prohibited as long as `DATA1` is not deleted. In these cases, the generalized driver was aborting with a message of the form:

  ```
  DRVDEP: THE MODIFICATION OF DATA3 BREAKS A PREVIOUS DEPENDENCY RULE.
  ```

  However, a copy of `DATA3` can be performed by the command

  ```
  DATA4 := DATA3 ;
  ```

and `DATA4` can be modified without restriction. Another possibility was to free `DATA3` from its dependencies by the command

```
DATA3 := FREE: DATA3 ;
```

In this new version, these dependency rules are no longer imposed. The main reason for this upgrade is that most loops and procedures (where object names do not usually change) include a large number of free commands that did not contribute to calculations. Moreover, some users were freeing every object after every module, making the syntax ridiculous.

- A linked list opened in `create` or `modification` mode is closed with its active directory recorded. This means that it will re-open on the directory which was active at the previous close.

  A linked list open in `read-only` mode is automatically closed on the directory active at the time of the open. This means that a linked list opened in `read-only` mode is left unchanged regardless of the operation performed on it.

- The calling sentence to a module should always end by a ";". A comment can follow on the same input data record but a carriage return should be performed before other significant data can be read by `REDGET`.

The possibility of user-defined procedures is also being offered. These procedures give the user the possibility to "program" an application using the capabilities of the generalized driver and to use it as a new module in the main data stream or in a calling procedure. A procedure is a set of input data specifications contained in a distinct ASCII file and called from another input data file. *The name of the procedure should be the same as the name of the* ASCII *file.* A procedure usually begins with the `PARAMETER` statement to define the interface between the calling and the called procedure. The correspondence of linked list or interface file parameters follows the rules previously described in the `PARAMETER` declaration section. For example, consider a procedure call of the form:

```
PROCEDURE    PROC ;
```

```
SEQ_ASCII    OBJ1 ;
LINKED_LIST  OBJ2 OBJ3 OBJ4 ;
...
OBJ1 OBJ2 := PROC OBJ2 OBJ3 OBJ4  ;
```

The procedure `PROC` is in fact a file named `PROC.c2m` and it could begin by a `PARAMETER` declaration of the short form:

```
PARAMETER XXX1 XXX2 XXX3 XXX4 ;
```

where `XXX1`, `XXX2`, `XXX3` and `XXX4` will be associated with `OBJ1`, `OBJ2`, `OBJ3` and `OBJ4` respectively. Using the long form, the user could also impose the type of objects he expects from the calling program:

```
PARAMETER XXX1 XXX2 XXX3 XXX4 ::
      ::: LINKED_LIST XXX2 XXX3 XXX4 ;
      ::: SEQ_ASCII   XXX1 ;
      ;
```

However, in this case, there must be a perfect match between types in order that the procedure can be executed.

3 The kernel drivers (programmer's guide)

A scientific application based on the GAN generalized driver is built by linking application-dependent modules to standard subroutines found in the GANLIB library. An application-dependent module recovers information from and stores information into the linked list using a tool-box of subroutines also found in the GANLIB library.

3.1 Developer's basic directives

The developer of any scientific application built around the GAN generalized driver should provide a main program of the following form (here given in Fortran-77):

```
      PROGRAM          GANTST
      IMPLICIT         NONE
      INTEGER          NMODXT,   MAXENT
      PARAMETER      ( NMODXT=1, MAXENT=64 )
      CHARACTER*12     CMODXT(NMODXT),HENTRY(MAXENT)
      INTEGER          IENTRY(MAXENT),JENTRY(MAXENT),KENTRY(MAXENT)
*
* LOCAL STORAGE
      INTEGER          IMODUL,NENTRY,ILEVEL,IPRINT
*
* GAN-2000 PARAMETERS
      INTEGER          CLECST
      EXTERNAL         CLECST
      INTEGER          KERNEL
*
* BLANK COMMON FOR DYNAMIC ALLOCATION IN F-77
      INTEGER          IBASE(1)
      REAL             RBASE
      COMMON           RBASE(1)
      EQUIVALENCE     (RBASE(1),IBASE(1))
*
      DATA CMODXT / 'MYTEST:     ' /
*
      ILEVEL= 0
      IPRINT= 0
   10 CONTINUE
         IMODUL= KERNEL(CMODXT,NMODXT,ILEVEL,MAXENT,
     1                  CLECST,IPRINT,
     2                  NENTRY,HENTRY,IENTRY,JENTRY,KENTRY)
*
*        IS THE RUN FINISHED ?
         IF( ILEVEL.EQ.0 ) GO TO 666
*
*        NO -> CALL THE APPROPRIATE CODE MODULE
         IF( IMODUL.EQ. 1 )THEN
            CALL MYTEST(NENTRY,HENTRY,IENTRY,JENTRY,KENTRY)
         ELSE
            WRITE(6,*) 'KERNEL ERROR CODE=',IMODUL
            CALL XABORT('GANTST: KERNEL ERROR')
         ENDIF
      GO TO 10
  666 STOP
      END
```

Here made of only one custom module named `MYTEST:`, this application would require a custom `MYTEST` subroutine defined by the developer to allow the linking with the generalized driver. If next calling statement makes sense, the kernel will send the module number on return; otherwise, the kernel sends an error code after some (generally unpleasant) output on errors. The form of every module is invariant:

```
      SUBROUTINE MYTEST(NENTRY,HENTRY,IENTRY,JENTRY,KENTRY)
*
*----------------------------------------------------------------------
*
* MYTEST MODULE. CALLED FROM THE MAIN PROGRAM.
*
* INPUT/OUTPUT PARAMETERS:
*  NENTRY : NUMBER OF LINKED LISTS AND FILES USED BY THE MODULE.
*  HENTRY : CHARACTER*12 NAME OF EACH LINKED LIST OR FILE.
*  IENTRY : =1 LINKED LIST;
*           =2 XSM FILE;
*           =3 SEQUENTIAL BINARY FILE;
*           =4 SEQUENTIAL ASCII FILE;
*           =5 DIRECT ACCESS FILE.
*  JENTRY : =0 THE LINKED LIST OR FILE IS CREATED.
*           =1 THE LINKED LIST OR FILE IS OPEN FOR MODIFICATIONS;
*           =2 THE LINKED LIST OR FILE IS OPEN IN READ-ONLY MODE.
*  KENTRY : FILE UNIT NUMBER OR LINKED LIST ADDRESS.
*           DIMENSION HENTRY(NENTRY),IENTRY(NENTRY),JENTRY(NENTRY),
*           KENTRY(NENTRY)
*
*------------------------------------ AUTHOR: R. ROY --- 23/03/00 ----
*
      IMPLICIT  NONE
      INTEGER   NENTRY
      CHARACTER HENTRY(*)*12
      INTEGER   IENTRY(*),JENTRY(*),KENTRY(*)
*
*     MODULE IS ALLOWED TO DO THINGS ON ENTRIES WITH JENTRY(.) <= 1
      ...
      RETURN
      END
```

Note that the module names are given in a list of available modules so that the compiler knows if the module is available or not. The module names do not have to be exactly the same as the module names, this is just a standard habit. Finally, at least one subroutine is provided for each module and the load module of the scientific application is linked using the operating system link edit. Note that the calling parameters of each module are similar. `NENTRY` is a single integer number whereas the other four are vectors of dimension `NENTRY`. They are defined in the following way:

`NENTRY` : Number of linked lists or files used by the module.

`HENTRY` : `character*12` name of each linked list or file.

`IENTRY` : Type of data structure:
=1 linked list;
=2 XSM file;
=3 sequential binary file;
=4 sequential ASCII file;
=5 direct access file.

`JENTRY` : Mode for the data structure:
=0 the linked list or file is created in the module;
=1 the linked list or file is opened for modifications in the module;
=2 the linked list or file is opened in read-only mode.

`KENTRY` : File unit number or linked list address.

The following developer's directives are always followed by an application built around the generalized driver:

- All the linked lists or files used to interface an application-dependent module are created, opened, closed and/or destroyed by the generalized driver.
- The programmer should not try to modify any entry opened in read-only mode by the generalized driver.
- The programmer should not try to open or close any entry used or managed by the generalized driver.
- The use of some subroutines of the GANLIB-2 library should be avoided in applications builded around the generalized driver. For example, we do not recommend the use of subroutines `XSM___`, `LCMOP` and `LCMCL`.
- The programmer should not never use the `STOP` statement. A call to the `XABORT` subroutine of the GANLIB-2 library should be used to stop the execution of a module.
- The programmer should not use any common block or any `INCLUDE` statement in its application-dependent modules, with the exception of the blank common `COMMON BASE(1)` used to locate the dynamically allocated memory in FORTRAN-77.

- All the input data required by a module *should be read by the free format input reader* `REDGET`. The module should read all the data up to (and including) the ";" character. All output data for CLE-2000 variables *should be written using* `REDPUT`.

- A pointer to the information declared as `LINKED_LIST` and `XSM_FILE` or a unit file number to the information declared as `SEQ_BINARY`, `SEQ_ASCII` and `DIR_ACCESS` is transmitted in the `KENTRY(I)` vector. The value of `I` depends on the position of the driver's variables when the call to the module is performed. For example, calling a module `MOD1:` with the following command:

  ```
  DATA1 DATA2 := MOD1: DATA2 DATA3 ;
  ```

  will associate index `I=1` with `DATA1`, index `I=2` with `DATA2` and index `I=3` with `DATA3`.

3.2 Kernel's tree

The basic access for any main program using the GANLIB is called KERNEL. From this kernel, the tree of routine calls is given in the following list:

```
kernel
|  lcmset
|  kdropn
|  clepil
|  |  clelog
|  |  clestk
|  |  clexrf
|  objpil
|  |  objstk
|  |  objxrf
|  lcmop
|  redopn
|  lcmsix
|  lcmget
|  lcmcar
|  lcmcl
|  kdrcls
|  redget
|  lcmput
|  redcls
|  drviox
|  |  redget
|  |  lcmput
|  |  redcls
|  |  lcmsix
|  |  lcmcar
|  |  xabort
|  |  lcmget
|  |  redput
|  |  redopn
|  drvequ
|  |  redget
|  |  xabort
|  |  kdropn
|  |  lcmexp
|  |  kdrcls
|  drvutl
|  |  xabort
|  |  lcmnxt
|  |  redget
|  |  lcmlen
|  |  setara
|  |  lcmget
|  |  rlsara
|  |  lcmsix
|  |  kdropn
|  |  lcmexp
|  |  kdrcls
|  |  lcmput
|  |  lcmop
|  |  lcmadd
|  |  lcmcl
|  |  lcmult
|  |  lcmlib
|  drvadd
```

```
|  |  xabort
|  |  kdropn
|  |  lcmexp
|  |  kdrcls
|  |  lcmadd
|  drvmpx
|  |  xabort
|  |  kdropn
|  |  lcmexp
|  |  kdrcls
|  |  redget
|  |  lcmult
|  drvsta
|  |  xabort
|  |  lcmsta
|  |  |  lcmval
|  |  |  lcmnxt
|  |  |  kdropn
|  |  |  xabort
|  |  |  lcmexp
|  |  |  lcmsix
|  |  |  lcmlen
|  |  |  setara
|  |  |  lcmget
|  |  |  rlsara
|  |  |  kdrcls
|  drvbac
|  |  xabort
|  |  redget
|  |  lcmsix
|  |  lcmequ
|  drvrec
|  |  redget
|  |  xabort
|  |  lcmlen
|  |  lcmsix
|  |  lcmequ
|  drvgrp
|  |  xabort
|  |  redget
|  |  lcmsix
|  |  lcmlen
|  |  setara
|  |  lcmget
|  |  redput
|  |  rlsara
|  drv000
|  |  xabort
|  |  redget
|  |  lcmsix
|  |  lcmput
|  |  lcmget
|  |  redput
|  kdrprm
|  |  lcmsix
|  |  lcmget
|  |  lcmcar
|  |  lcmput
|  |  redget
|  |  redcls
|  |  kdrdmd
|  |  |  redget
```

```
|  |  |  lcmsix
|  |  |  lcmget
|  |  kdrdll
|  |  |  redget
|  |  |  lcmsix
|  |  |  lcmget
|  |  kdrdxf
|  |  |  redget
|  |  |  lcmsix
|  |  |  lcmget
|  |  kdrdsb
|  |  |  redget
|  |  |  lcmsix
|  |  |  lcmget
|  |  |  lcmput
|  |  |  lcmcar
|  |  kdrdsa
|  |  |  redget
|  |  |  lcmsix
|  |  |  lcmget
|  |  |  lcmput
|  |  |  lcmcar
|  |  kdrdda
|  |  |  redget
|  |  |  lcmsix
|  |  |  lcmget
|  |  |  lcmput
|  |  |  lcmcar
|  |  redopn
|  kdrdpr
|  |  redget
|  |  lcmsix
|  |  lcmget
|  |  lcmcar
|  |  kdropn
|  |  clepil
|  |  |  clelog
|  |  |  clestk
|  |  |  clexrf
|  |  objpil
|  |  |  objstk
|  |  |  objxrf
|  |  kdrcls
|  |  lcmput
|  kdrdmd
|  |  redget
|  |  lcmsix
|  |  lcmget
|  kdrdll
|  |  redget
|  |  lcmsix
|  |  lcmget
|  kdrdxf
|  |  redget
|  |  lcmsix
|  |  lcmget
|  kdrdsb
|  |  redget
|  |  lcmsix
|  |  lcmget
|  |  lcmput
|  |  lcmcar
```

```
|  kdrdsa
|  |  redget
|  |  lcmsix
|  |  lcmget
|  |  lcmput
|  |  lcmcar
|  kdrdda
|  |  redget
|  |  lcmsix
|  |  lcmget
|  |  lcmput
|  |  lcmcar
|  clecop
|  kdrcln
|  |  lcmnxt
|  |  lcmsix
|  |  lcmget
|  |  lcmcar
|  |  lcmop
|  |  lcmcl
|  |  kdropn
|  |  kdrcls
|  |  lcmput
|  kdrfst
```

3.3 The KERNEL function

The main routine provided by the generalized driver is its kernel. The kernel is responsible for driving the application and for giving some error messages in the case something goes wrong.

The calling syntax for the kernel is the following:

```
IMODUL= KERNEL(CMODXT,NMODXT,ILEVEL,MAXENT,
               CLECST,IPRINT,
               NENTRY,HENTRY,IENTRY,JENTRY,KENTRY)
```

input parameters:

`CMODXT` : `character*12` vector containing the module names of the application.

`NMODXT` : number of modules in the application.

`ILEVEL` : current computational level. It must be set to 0 as beginning. It is changed inside the kernel.

`MAXENT` : maximum number of linked lists or files allowed in a single module call.
Suggested value: between 8 and 64.

`CLECST` : external routine proving application-dependent parameters to be used in the CLE-2000 compiler. i Default routine has this name.

`IPRINT` : printing level for the kernel.
Suggested value: 0.

ouput parameters:

`IMODUL` : current number of the application module to be processed if positive. If zero or negative, there was a kernel error.

`ILEVEL` : current computational level. The kernel will add one to this value each time a new source file is processed, subtract one each time the source file directives are finished.

`NENTRY` : Number of linked lists or files to be used by the next module.

`HENTRY` : `character*12` name of each linked list or file.

`IENTRY` : Type of data structure:
=1 linked list;
=2 XSM file;
=3 sequential binary file;
=4 sequential ASCII file;
=5 direct access file.

`JENTRY` : Mode of the data structure:
=0 the linked list or file is created in the module.
=1 the linked list or file is opened for modifications in the module;
=2 the linked list or file is opened in read-only mode.

`KENTRY` : File unit number or linked list address.

These values (except possibly the returned value `IMODUL`) **should not be modified** by any module, because these are needed by the kernel on its next call. As explained above, an application based on this kernel is an infinite loop that:

calls the `KERNEL` function, recovers the module number `IMODUL` ;

launches the module `IMODUL`, waits that it ends;

this proceeds until it ends with `ILEVEL=0` or crashes (the final result generally depending on user's experience).

The kernel uses the CLE-2000 compiler's routines[4] which will not be described here. However, the kernel also uses some extensions that we will now describe:

3.4 Input probing

It is not possible to detect any mistake that can happen with user's input. A minimal job is done using the `OBJPIL` tools; the idea is to detect errors of the following types:

a module, procedure, linked list or file was used before being declared;

a module or procedure is not available in this application or calculation.

3.4.1 OBJPIL

For each procedure to be compiled (including the main user's input), the kernel calls the following integer function which has 4 input arguments and 1 output argument:

```
IRETCD=OBJPIL(IUNITO,IWRITE,CMODXT,NMODXT)
```

input parameters:

`IUNITO` : unit number of the CLE-2000 direct-access file. Usually, the file name has the extension `*.o2m`.

`IWRITE` : unit number where comments will be printed.

`CMODXT` : `character*12` vector containing the module names of the application.

`NMODXT` : number of modules in the application.

The output value `IRETCD` is zero when no error was encountered when compiling the file. The **OBJPIL** is a concatenation of 2 successive routine calls to **OBJSTK** and **OBJXRF** whose functions will now be explained.

3.4.2 OBJSTK

Call used to set up a list of declared names and check consistence.

```
IRETCD=OBJSTK(IUNITO,IWRITE,CMODXT,NMODXT)
```

The arguments are the same as in **OBJPIL** . This routine is called by **OBJPIL** , and should therefore be never directly used by developers.

3.4.3 OBJXRF

Call used to perform cross-reference of linked lists and files from data of the source file.

```
IRETCD=OBJXRF(IUNITO,IWRITE)
```

This cross-reference is printed onto `IWRITE` unit when the user ask for it in the `QUIT` statement. The arguments have the same meaning as in **OBJPIL** . This routine is called by **OBJPIL** , and should therefore be never directly used by developers.

3.5 File management

3.5.1 KDROPN

Command used to allocate a file unit associated to a given file name. Sequential (ASCII or binary) and direct access (XSM or not) files are permitted. The generalized driver is responsible for making all the **KDROPN** calls required to allocate the interface files used by a module. The only use of this call is therefore to allocate a temporary (i.e., scratch) file from inside a module.

Every file used by an application based on the generalized driver should be allocated through **KDROPN** .

```
IUNIT=KDROPN(CUNAME,IACTIO,IUTYPE,LRDA,IHANDL)
```

input parameters:

`CUNAME` : `character*72` name of the file we want to allocate.

`IACTIO` : action type:
=0 to allocate a new file;
=1 to access and modify an existing file;
=2 to access an existing file in **read-only** mode.

`IUTYPE` : file type:
=1 XSM file;
=2 sequential unformatted;
=3 sequential formatted (i.e., ASCII file); =4 direct access (DA) unformatted file.

`LRDA` : number of single words in a DA record (given with `IUTYPE` equal to 4 only; otherwise set to zero).

`IHANDL` : pointer to the XSM file handle (given with `IUTYPE` equal to 1 only; otherwise set to zero).

ouput parameter:

`IUNIT` : unit number or error status value:
>0 unit number allocated;

=-1 no more units available;

=-2 file type requested inconsistent with existing type;

=-3 file already opened;

=-4 `CUNAME` is a reserved name;

=-5 illegal value of `IUTYPE`;

=-6 error on open of XSM file;

=-7 error on open of unformatted sequential file;

=-8 error on open of formatted sequential (i.e., ASCII) file;

=-9 error on open of DA file; =-10 illegal value of `LRDA`;

=-11 DA record length (`LRDA`) inconsistent with existing length.

3.5.2 KDRCLS

Command used to close and release a file unit. Sequential (ASCII or binary) and direct access (XSM or not) files are permitted. The generalized driver is responsible for making all the **KDRCLS** calls required to close and release the interface files used by a module. The only use of this call is therefore to close and release a temporary (i.e., scratch) file from inside a module.

Every file used by an application based on the generalized driver should be closed and released through **KDRCLS** .

```
IERR=KDRCLS(ITAPNO,IACTIO)
```

input parameters:

`ITAPNO` : unit number:

=0 to close all the open units;

>0 unit to close.

`IACTIO` : action type:

=1 to keep the file;

=2 to delete the file.

ouput parameter:

`IERR` : completion code:

=0 the unit is successfully closed;

=-2 the unit was not previously opened;

=-3 the unit has been opened by routines other than KDROPN;

=-4 file unit is reserved (5,6);

=-5 illegal unit number;

=-6 error on close of XSM file;

=-7 error on close of unformatted sequential file;

=-8 error on close of formatted sequential (i.e., ASCII) file);

=-9 error on close of DA file; =-10 illegal value of IACTIO.

3.5.3 KDRFST

Command used to return the unit allocated to file with name CUNAME or the file name allocated to unit ITAPNO. Also it returns the type IUTYPE of that file, the record length LRDA if IUTYPE=4, and the pointer IHANDL to the XSM file handle if IUTYPE=1.

```
IERR=KDRFST(CUNAME,ITAPNO,LRDA,IHANDL)
```

input parameters:

CUNAME : blank character (' ') or character*72 name of the file for which we want the unit number.

ITAPNO : unit number of the file for which we want the character*72 name.

ouput parameters:

CUNAME : character*72 name of the file (if CUNAME=' ' at input).

ITAPNO : unit number of the file (if CUNAME is defined at input).

LRDA : number of single words in a DA record (given with IUTYPE equal to 4 only; otherwise set to zero).

IHANDL : address of the XSM file handle (given with IUTYPE equal to 1 only; otherwise set to zero).

IERR : completion code:

>0 value of IUTYPE;

=-2 the unit was not previously opened;

=-3 the unit has been opened by routines other than `KDROPN`;

=-4 file unit is reserved (5,6);

=-5 illegal unit number.

3.6 The LCM software

The programmer of a scientific application needs to know only a subset of the GANLIB library to work with linked lists or files. For example, record `RECORD8` of the linked list represented in Fig. 1 can be recovered from within an application-dependent module using the following toolbox calls:

```
CALL LCMSIX(KENTRY(I),'DIRECTORY1',1)
   CALL LCMSIX(KENTRY(I),'DIRECTORY3',1)
   CALL LCMLEN(KENTRY(I),'RECORD8',ILENGT,ITYLCM)
   CALL LCMGET(KENTRY(I),'RECORD8',DATA)
   CALL LCMSIX(KENTRY(I),' ',2)
CALL LCMSIX(KENTRY(I),' ',2)
```

where calls to `LCMSIX` are used to move in the hierarchical structure of the linked list whose address is `KENTRY(I)`. `LCMLEN` permits the length (`ILENGT`) and type (`ITYLCM`) of the record `RECORD8` to be recovered. Finally, `LCMGET` permits the record `RECORD8` to be loaded into a vector `DATA(I)` of dimension greater than or equal to `ILENGT`.

Each linked list or XSM file is referred by an integer variable `IPLIST` containing the address of its handle. A linked list keeps the same `IPLIST` throughout its existence entire whereas an XSM file uses a different `IPLIST` each time it is reopened.

The detailed specification of these calls will now be presented.

3.6.1 LCMSET

These routines are used to translate character variables into integer vectors back and forth, The routine **LCMSET** is portable and base on the ASCII collating sequence. The **KERNEL** routine initializes this translation by calling:

```
CALL LCMSET
```

This sets up a look-up table that can then be used.

3.6.2 LCMCAR

Whenever a developer wants to do a character/integer conversion, he should call:

```
CALL LCMCAR(TEXT,LACTIO,NITMA)
```

input/output parameters:

`LACTIO` : logical value if `.TRUE.`, then translation from character to integer. Otherwise, translation from integer to character.

`TEXT` : character variable whose length **must** be a multiple of 4.

`NITMA` : integer vector whose dimension agrees with `LEN(TEXT)/4`.

The **LCMCAR** translator was found to be faster than the usual internal read of FORTRAN. In all cases, when exporting data onto a sequential ASCII file, the characters are always visible (as long as developers have used the character type in their **LCMPUT** calls).

3.6.3 LCMOP

Call used to open a linked list (or an XSM file). The generalized driver is responsible for making all the **LCMOP** calls required to open the linked lists and XSM files used by a module. The only use of this call is therefore to open a temporary (i.e., scratch) linked list (or XSM file) from inside a module.

```
CALL LCMOP(IPLIST,NAMP,IMP,MEDIUM,IMPX)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of a XSM file if `IMP=1` or `IMP=2`.

`NAMP` : character*12 name of the linked list (or XSM file) if `IMP=0`.

`IMP` : opening mode:
=0 to create a new linked list (or XSM file);
=1 to modify an existing linked list (or XSM file);
=2 to access a linked list (or XSM file) in **read-only** mode.

`MEDIUM` : kind of medium:
=1 use a linked list;
=2 use an XSM file.

`IMPX` : print parameter. Equal to zero to suppress printing.

output parameters:

`IPLIST` : address of a linked list or address of the handle of a XSM file if `IMP=0`.

`NAMP` : `character*12` name of the linked list (or XSM file) if `IMP=1` or `IMP=2`.

3.6.4 LCMLEN

Command used to recover the length and type of a directory or a block of information stored in the active directory of a linked list (or in the active directory of an XSM file). The value of the length recovered is the number of single words required to store the information. For example, the length of a `DOUBLE PRECISION` array is twice its dimension.

```
CALL LCMLEN(IPLIST,NAMP,ILONG,ITYLCM)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`NAMP` : `character*12` name of the block or directory.

output parameters:

`ILONG` : length in single words of the block.
=-1 is returned for a directory;
=0 is returned if the block or directory does not exist.

`ITYLCM` : Type of information:
=0 directory;
=1 integer;
=2 single precision real;
=3 `character*4` data;
=4 double precision real;
=5 logical;
=6 complex number;
=7 undefined (7 is returned if the block or directory does not exist.

3.6.5 LCMGET

Command used to recover a block of information stored in the active directory of a linked list (or in the active directory of an XSM file) and copy it into core memory.

```
CALL LCMGET(IPLIST,NAMP,DATA)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle to a XSM file.

`NAMP` : `character*12` name of the block to recover. An abort is performed if this block does not exist.

output parameters:

`DATA` : vector of dimension at least `ILONG` into which the block has been copied.

The subroutine **LCMGET** can be used to recover character data written in a block. There are two ways to do such a thing. Firstly, a block of length 5 named `NAMP` stored in the active directory of the linked list (or XSM file) pointed to by `IPLIST` is recovered using **LCMGET** and copied into the `character*20` variable named `HNAME` using an internal `WRITE` statement:

```
CHARACTER NAMP*12,HNAME*20
INTEGER IDATA(5)
IPLIST=...
NAMP=...
CALL LCMGET(IPLIST,NAMP,IDATA)
WRITE(HNAME,'(5A4)') (IDATA(I),I=1,5)
```

Secondly, it is also possible to use the **LCMCAR** routine. In that case, the same code looks like:

```
CHARACTER NAMP*12,HNAME*20
INTEGER IDATA(5)
LOGICAL LACTIO
IPLIST=...
NAMP=...
CALL LCMGET(IPLIST,NAMP,IDATA)
LACTIO=.FALSE.
CALL LCMCAR(HNAME,LACTIO,IDATA)
```

It is not recommended to mix both ways of doing conversion inside the same module because this may product side effects.

3.6.6 LCMPUT

Command used to store a block of information in the active directory of a linked list (or in the active directory of an XSM file). The information is copied from the core memory into the linked list. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed on a linked list or XSM file open in `read-only` mode.

```
CALL LCMPUT(IPLIST,NAMP,ILONG,ITYLCM,DATA)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`NAMP` : `character*12` name of the block.

`ILONG` : length in single words of the block. If a block contain N double precision values, `ILONG` should be set to $2 \times N$.

`ITYLCM` : Type of information:
=1 integer;
=2 single precision real;
=3 `character*4` data;
=4 double precision real;
=5 logical;
=6 complex number;
=7 undefined.

`DATA` : vector of dimension at least `ILONG` from which the block is copied. The first `ILONG` elements of `DATA` should be properly initialized before the call to `LCMPUT` is performed.

The subroutine **LCMPUT** can be used to store character data in a block. Once more, there are two ways of doing it. Firstly, a `character*20` variable named `HNAME` will be copied into the integer array of length 5 named `IDATA` using an internal `READ` statement and stored as a block named `NAMP` in the active directory of the linked list (or XSM file) pointed to by `IPLIST` using **LCMPUT** :

```
CHARACTER NAMP*12,HNAME*20
INTEGER IDATA(5)
IPLIST=...
NAMP=...
READ(HNAME,'(5A4)') (IDATA(I),I=1,5)
CALL LCMPUT(IPLIST,NAMP,5,3,IDATA)
```

Secondly, using the **LCMCAR** routine, the same code looks like:

```
CHARACTER NAMP*12,HNAME*20
INTEGER IDATA(5)
LOGICAL LACTIO
```

```
IPLIST=...
NAMP=...
LACTIO=.TRUE.
CALL LCMCAR(HNAME,LACTIO,IDATA)
CALL LCMPUT(IPLIST,NAMP,5,3,IDATA)
```

It is not recommended to mix both ways of doing conversion inside the same module because this may product side effects. If the combination **LCMCAR /LCMPUT** was used, then it is expected that values will be recovered using the combination **LCMGET /LCMCAR** .

3.6.7 LCMSIX

Command used to move in the hierarchical structure of a linked list (or in the the hierarchical structure of an XSM file) or to change the active directory. If the linked list or XSM file is open in `read-only` mode, a move to a non-existent directory cannot be performed.

```
CALL LCMSIX(IPLIST,NAMP,IACT)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`NAMP` : `character*12` name of the daughter directory if `IACT=1`. This parameter is not used if `IACT=0` or `IACT=2`.

`IACT` : Type of movement.
=0 move back to the root directory;
=1 move to a daughter directory named `NAMP` (create it if it does not already exist);
=2 move back to the parent directory.

3.6.8 LCMNXT

Command used to find the name of the next block or directory in the active directory of a linked list (or in the active directory of an XSM file).

```
CALL LCMNXT(IPLIST,NAMP,NAMLCM,NAMMY,LCM)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`NAMP` : `character*12` name of an existing block or directory. `NAMP=' '` can be used at input to obtain the name of the first block or directory.

output parameters:

`NAMP` : `character*12` name of the next block or directory. Returned as `NAMP=' '` if the directory is empty.

`NAMLCM` : `character*12` name of the linked list or XSM file.

`NAMMY` : `character*12` name of the active directory.

`LCM` : logical variable set to `.TRUE.` if the information is stored in a linked-list or set to `.FALSE.` if the information is stored in an XSM file.

3.6.9 LCMIOF

Command used to recover a **SETARA** offset to a block of information stored in the active directory of a linked list *without making a copy* of the information.

A call to **LCMIOF** *can only be performed if the information is stored in a linked list (i.e., if* `LCM` *is* `.true.`*). If the information is modified, a call to* **LCMPOF** *should be subsequently performed to acknowledge the modifications.*

This function represents an advanced capability of the `LCM` software and it is only intended to be used in situations where the reduction of CPU resources is a primary issue. An offset is returned in a vector `BASE` stored in a common block of the form `COMMON BASE(1)`. A call to `LCMIOF` does not cause any modification to the linked list. The useful information is therefore found from `BASE(IOFDUM) to BASE(IOFDUM+ILONG-1)`. *The programmer of a scientific application should never try to release* `BASE(IOFDUM)`.

```
IOFDUM=LCMIOF(IPLIST,NAMP)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`NAMP` : `character*12` name of the block to recover. An abort is performed if this block does not exist.

ouput parameters:

`IOFDUM` : `SETARA` offset of the information.

3.6.10 LCMPOF

Command used to store a block of information in the active directory of a linked list (or in the active directory of an XSM file) *without making a copy* of the information. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed on a linked list or XSM file that is open in `read-only` mode.

If the entry `NAMP` *already exists, its* **SETARA** *pointer is replaced by the new one in the LCM database and the information pointed to by the old* **SETARA** *pointer is deallocated.*

This function represents an advanced capability of the `LCM` software and it is only intended to be used in situations where the reduction of CPU resources is a primary issue. The information stored by `LCMPOF` should have been previously allocated by a call to `SETARA` of the form `CALL SETARA(BASE,ILONG,IOFDUM)` where `BASE` is a vector stored in a common block of the form `COMMON BASE(1)`.

```
CALL LCMPOF(IPLIST,NAMP,ILONG,ITYLCM,IOFDUM)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`NAMP` : `character*12` name of the block.

`ILONG` : length in single words of the block.

`ITYLCM` : Type of information:
=1 integer;
=2 single precision real;
=3 `character*4` data;
=4 double precision real;
=5 logical;
=6 complex number;
=7 undefined.

`IOFDUM` : `SETARA` offset to the information. The elements of `BASE(IOFDUM)` to `BASE(IOFDUM+ILONG-1)` should be properly initialized before the call to `LCMPOF` is performed.

ouput parameters:

`IOFDUM` : `IOFDUM=0` at output to indicate that the information previously pointed by `IOFDUM` is now managed by the `LCM` software.

3.6.11 LCMDEL

Command used to delete a block of information stored in the active directory of a linked list. The **LCMDEL** capability is not available with an XSM file.

```
CALL LCMDEL(IPLIST,NAMP)
```

input parameters:

`IPLIST` : address of a linked list.

`NAMP` : `character*12` name of the block to delete.

3.6.12 LCMLIB

Command used to print the content of the active directory of a linked list (or the active directory of an XSM file).

```
CALL LCMLIB(IPLIST)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

3.6.13 LCMADD

Command used to add the floating point information contained in the active and daughter directories of two linked lists (or the active directories of two XSM files). The result is stored in the linked list or XSM file pointed to by `IPLIS2`.

```
CALL LCMADD(IPLIS1,IPLIS2)
```

input parameters:

`IPLIS1` : address of the first linked list or address of the handle of the first XSM file.

`IPLIS2` : address of the second linked list or address of the handle of the second XSM file.

ouput parameter:

`IPLIS2` : address of the linked list or address of the handle of the XSM file containing the result of the addition.

3.6.14 LCMULT

Command used to multiply the floating point information contained in the active and daughter directories of a linked list (or the active directories of an XSM file) by a real number `FLOTT`. The result is stored in the linked list or XSM file pointed to by `IPLIST`.

```
CALL LCMULT(IPLIST,FLOTT)
```

input parameter:

`IPLIST` : address of the linked list or address of the handle of the XSM file.

`FLOTT` : real number used to multiply the linked list or the XSM file.

ouput parameter:

`IPLIST` : address of the linked list or address of the handle of the XSM file containing the result of the multiplication.

3.6.15 LCMEQU

Command used to copy the information contained in the active and daughter directories of the linked list or XSM file pointed to by `IPLIS1` into the linked list or XSM file pointed to by `IPLIS2`. Note that the second linked list (or XSM file) is modified but not created by **LCMEQU** .

```
CALL LCMEQU(IPLIS1,IPLIS2)
```

input parameters:

`IPLIS1` : address of the existing linked list or address of the handle of the existing XSM file (accessed in read-only mode).

`IPLIS2` : address of the linked list or address of the handle of the XSM file that will be modified by `LCMEQU`.

ouput parameter:

`IPLIS2` : address of the linked list or address of the handle of the XSM file modified by `LCMEQU`.

3.6.16 LCMSTA

Command used to compare the floating point information contained in the active and daughter directories of two linked lists (or the active directories of two XSM files). The absolute maximum and averaged differences are printed on the output listing.

```
CALL LCMSTA(IPLIS1,IPLIS2)
```

input parameters:

`IPLIS1` : address of the first linked list or address of the handle of the first XSM file.

`IPLIS2` : address of the second linked list or address of the handle of the second XSM file.

3.6.17 LCMEXP

Command used to export (import) the content of a linked list or XSM file to (from) a sequential binary or ASCII file using the contour method. The export starts from the active directory.

```
CALL LCMEXP(IPLIST,IMPX,NUNIT,IMODE,IDIR)
```

input parameters:

`IPLIST` : address of the linked list or address of the handle of the XSM file that is to be exported (or imported).

`IMPX` : print parameter (set to zero for no print).

`NUNIT` : unit number of the sequential file where the export is written (or from which the import is recovered).

`IMODE` : status mode:
=1 for sequential binary export (import) file;
=2 for sequential ASCII export (import) file.

`IDIR` : directive:
=1 to export;
=2 to import.

3.6.18 LCMVAL

Command used to validate a single entry or the complete active and daughter directories of a linked list. If `IPLIST` refers to an XSM file, no action is performed. The validation consists of checking the connections between the elements of the linked list, checking if every element of the linked list is defined and checking possible memory overwriting on the linked list. If something wrong is detected, the following message is issued:

```
LCMVAL: BLOCK xxx OF THE LINKED LIST yyy HAVE BEEN OVERWRITTEN.
```

The calling specification is:

```
CALL LCMVAL(IPLIST,NAMP)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`NAMP` : `character*12` name of the block to be validated in the linked list. If `NAMP=' '`, the active and daughter directories are validated.

3.6.19 LCMCL

Call used to close a linked list (or an XSM file). The generalized driver is responsible for making all the **LCMCL** calls required to close the linked lists and XSM files used by a module. The only use of this call is therefore to close a temporary (i.e., scratch) linked list (or XSM file) open from inside a module.

```
CALL LCMCL(IPLIST,IACT)
```

input parameters:

`IPLIST` : address of a linked list or address of the handle of an XSM file.

`IACT` : action to take
=1 close the linked list without destroying it;
=2 destroy it.

output parameters:

`IPLIST` : `IPLIST=0` at output to indicate that the linked list is destroyed or that the XSM file is closed. A linked list keep the same `IPLIST` during all its existence. An XSM file is handled by a different `IPLIST` each time it is re-opened.

3.7 Abort and exception handling

3.7.1 XABORT

Command used to perform a *clean abort* from inside a module. This subroutine perform the following functions:

- close all the files previously opened by **KDROPN** , taking a special care to save the XSM file buffers;
- print a message on the output listing;
- stop the execution.

This subroutine is mainly useful to process abnormal situations that can occurs in an application. It is recommended to include the name of the current subroutine or function in the `XABORT` message. For example, if something is going wrong in the `SUB001` subroutine, we may write:

```
CALL XABORT('SUB001: EXECUTION FAILURE.')
```

The detailed specification is:

`CALL XABORT(HSMG)`

input parameter:

`HSMG` : `character` message describing the abort conditions.

3.8 Dynamic allocation of memory in Fortran-77

This function allows FORTRAN-77 applications to dynamically allocate and release a zone of memory by calling memory management functions of the underlying operating system (**malloc** on UNIX computers). These capabilities are required in a modular application to adjust its core storage to the size of the problem being analyzed.

3.8.1 SETARA

Command used to allocate a zone of memory and return an offset in a vector `BASE` stored in a common block of the form `COMMON BASE(1)`. If the operating system is unable to allocate `ILONG` single words, an abort is performed.

```
CALL SETARA(BASE,ILONG,IOFDUM)
```

input parameters:

`BASE` : The address of `BASE(1)` is used as the origin by `SETARA` to compute `IOFDUM`.

`ILONG` : Length in single words of the allocated zone. To allocate a zone of memory to store N double precision words, `ILONG` should be set to $2 \times N$.

output parameters:

`IOFDUM` : Offset of the allocated zone in the `BASE` vector. The allocated zone is therefore found from `BASE(IOFDUM)` to `BASE(IOFDUM+ILONG-1)`.

Note that `BASE(1)` should never be declared as a double precision vector because the offset `IOFDUM` is computed by assuming that `BASE` contains only single words.

3.8.2 RLSARA

Command used to release a previously allocated zone of memory. If the operating system is unable to release the memory zone, an abort is performed.

```
CALL RLSARA(BASE(IOFDUM))
```

input parameters:

`BASE(IOFDUM)` : Offset in `BASE` of the first word of the memory zone to be released.

3.8.3 Example of memory allocation in FORTRAN-77

In the first example, a memory zone is dynamically allocated, a record `'RECORD1'` is created and copied into a linked list using the `LCMPUT` subroutine. A subroutine `SUB` is called to facilitate the creation of the record:

```
      COMMON BASE(1)
      .
      .
      .
      ILONG=30
      CALL SETARA(BASE,ILONG,IOFDUM)
      CALL SUB(BASE(IOFDUM),ILONG)
      CALL LCMPUT(IPLIST,'RECORD1',ILONG,2,BASE(IOFDUM))
      CALL RLSARA(BASE(IOFDUM))
      RETURN
      END
      .
      .
      .
      SUBROUTINE SUB(PHI,ILONG)
      DIMENSION PHI(ILONG)
      DO 10 I=1,ILONG
   10 PHI(I)=REAL(I)
      RETURN
      END
```

In the second example, a memory zone is dynamically allocated, a record `'RECORD1'` is created and stored in a linked list (without copying it) using the `LCMPOF` subroutine:

```
      COMMON BASE(1)
      .
      .
      .
      ILONG=30
      CALL SETARA(BASE,ILONG,IOFDUM)
      CALL SUB(BASE(IOFDUM),ILONG)
      CALL LCMPOF(IPLIST,'RECORD1',ILONG,2,IOFDUM)
      RETURN
      END
      .
      .
      .
      SUBROUTINE SUB(PHI,ILONG)
      DIMENSION PHI(ILONG)
      DO 10 I=1,ILONG
   10 PHI(I)=REAL(I)
      RETURN
      END
```

In the third example, a block stored in a linked list is recovered

- using the **LCMIOF** function, without copying it and without allocating memory (if `IPLIST` is a linked list);

- using the **LCMGET** subroutine and copied into a zone of memory dynamically allocated by SETARA (if IPLIST is an XSM file).

A subroutine SUB is called to facilitate the utilization of the recovered information:

```
COMMON BASE(1)
LOGICAL LCM
CHARACTER*12 NAMP,NAMLCM,NAMMY
.
.
.
NAMP=' '
CALL LCMNXT(IPLIST,NAMP,NAMLCM,NAMMY,LCM)
CALL LCMLEN(IPLIST,'RECORD1',ILONG,ITYLCM)
IF(LCM) THEN
   IOFDUM=LCMIOF(IPLIST,'RECORD1')
   CALL SUB(BASE(IOFDUM),ILONG)
ELSE
   CALL SETARA(BASE,ILONG,IOFDUM)
   CALL LCMGET(IPLIST,'RECORD1',BASE(IOFDUM))
   CALL SUB(BASE(IOFDUM),ILONG)
   CALL RLSARA(BASE(IOFDUM))
ENDIF
RETURN
END
.
.
.
SUBROUTINE SUB(PHI,ILONG)
DIMENSION PHI(ILONG)
PRINT *,'VECTOR PHI CONTENT=',(PHI(I),I=1,ILONG)
RETURN
END
```

4 CONCLUSION

The GAN generalized driver is not specific to lattice calculations and its use could be extended to other aspects of engineering. Linked lists and modules can be defined to handle full core diffusion calculations, fuel management and optimization, thermal-hydraulics, structural mechanics and metallurgy. This would enable these topics to be easily interfaced without programming a single line of FORTRAN. Finally, an expert system could be designed to manage the interaction between all the modules and databases involved in the complete design study of a nuclear reactor.

Acknowledgements

This work has been carried out partly with the help of grants from Hydro-Québec and the Natural Sciences and Engineering Research Council of Canada. We would like to thank my beta-testers, Guy Marleau and Siamak Kaveh for their help in clarifying some error messages issued by the driver.

References

[1] G. Marleau, A. Hébert and R. Roy, *A User's Guide for DRAGON,* Report IGE-174 Rev.3, Institut de Génie Nucléaire, École Polytechnique de Montréal, December 1997.

[2] E. Varin, A. Hébert, R. Roy and J. Koclas, *A User's Guide for DONJON,* Report IGE-208, Institut de Génie Nucléaire, École Polytechnique de Montréal, November 1996.

[3] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *"Numerical Recipes in FORTRAN: The Art of Scientific Computing. Second Edition,"* Cambridge U. P., Cambridge, ISBN 0-521-43064-X (1994).

[4] R. Roy, *The CLE-2000 Tool-box,* Report IGE-163, Institut de Génie Nucléaire, École Polytechnique de Montréal, December 1999.

Index