

TECHNICAL REPORT
IGE-332

**THE GANLIB5 KERNEL GUIDE
(64-BIT CLEAN VERSION)**

A. HÉBERT AND R. ROY



Institut de génie nucléaire
Département de génie mécanique
École Polytechnique de Montréal
May 4, 2016

Contents

Contents	ii
1 The GANLIB Version 5 architecture	1
1.1 From Versions 3 or 4 to Version 5	1
2 The ANSI C CLE-2000 API	3
2.1 The main entry point for CLE-2000	3
2.1.1 cle2000.c	3
2.1.2 dummod	3
2.1.3 Calling a main CLE-2000 procedure	4
2.1.4 Calling a parametrized CLE-2000 procedure	6
2.1.5 Calling a CLE-2000 procedure with in-out CLE-2000 variables	8
2.2 Calling a calculation module without a CLE-2000 procedure	9
2.2.1 clemod.c	9
2.3 Management of the last-in-first-out (lifo) stack	11
2.3.1 cleopn	12
2.3.2 clepop	13
2.3.3 clepush	13
2.3.4 clecls	13
2.3.5 clenode	13
2.3.6 clepos	14
2.3.7 clelib	14
2.4 The free-format input reader	14
2.4.1 redopn.c	14
2.4.2 redget.c	15
2.4.3 redput.c	15
2.4.4 redcls.c	16
2.5 Defining built-in constants in CLE-2000	16
2.5.1 clecst	16
3 The ANSI C LCM API	18
3.1 General utility functions	19
3.1.1 strcut.c	19
3.1.2 strfil.c	19
3.2 Opening, closing and validation of LCM objects	20
3.2.1 lcmop.c	20
3.2.2 lcmcl.c	20
3.2.3 lcmval.c	21
3.3 Interrogation of LCM objects	21
3.3.1 lcmllen.c	21
3.3.2 lcminf.c	22
3.3.3 lcmnxt.c	22
3.3.4 lcmlel.c	23
3.4 Management of the array of elementary type	23
3.4.1 lcmget.c	24
3.4.2 lcmput.c	25
3.4.3 lcmgpd.c	26
3.4.4 lcmppd.c	26
3.4.5 lcmdel.c	27
3.4.6 lcmgdl.c	27
3.4.7 lcmmdl.c	28
3.4.8 lcmgpl.c	29
3.4.9 lcmmpl.c	29

3.5	Management of the associative tables and of the heterogeneous lists	30
3.5.1	lcmdid_c	30
3.5.2	lcmlid_c	31
3.5.3	lcmlil_c	32
3.5.4	lcmdil_c	32
3.5.5	lcmgid_c	34
3.5.6	lcmgil_c	34
3.5.7	lcmsix_c	34
3.6	LCM utility functions	35
3.6.1	lcmlib_c	35
3.6.2	lcmequ_c	35
3.6.3	lcmexp_c	36
3.7	Using variable-length string arrays	36
3.7.1	lcmgcd_c	36
3.7.2	lcmpcd_c	37
3.7.3	lcmgcl_c	38
3.7.4	lcmpcl_c	38
3.8	Dynamic allocation of the elementary blocks of data	39
3.8.1	setara_c	39
3.8.2	rlsara_c	40
3.9	Abnormal termination of the execution	40
3.9.1	xabort_c	40
4	The ISO Fortran LCM API	41
4.1	Opening, closing and validation of LCM objects	41
4.1.1	LCMOP	41
4.1.2	LCMCL	41
4.1.3	LCMVAL	42
4.2	Interrogation of LCM objects	42
4.2.1	LCMLEN	43
4.2.2	LCMINF	43
4.2.3	LCMNXT	44
4.2.4	LCMLEL	44
4.3	Management of the array of elementary type	45
4.3.1	LCMGET	45
4.3.2	LCMPUT	46
4.3.3	LCMGPD	46
4.3.4	LCMPPD	47
4.3.5	LCMDEL	48
4.3.6	LCMGDL	48
4.3.7	LCMPDL	49
4.3.8	LCMGPL	50
4.3.9	LCMPPL	51
4.4	Management of the associative tables and of the heterogeneous lists	52
4.4.1	LCMDID	52
4.4.2	LCMLID	52
4.4.3	LCMLIL	53
4.4.4	LCMDIL	54
4.4.5	LCMGID	55
4.4.6	LCMGIL	56
4.4.7	LCMSIX	56
4.5	LCM utility functions	57
4.5.1	LCMLIB	57
4.5.2	LCMEQU	57

4.5.3	LCMEXP	57
4.6	Using fixed-length character arrays	58
4.6.1	LCMGTC	58
4.6.2	LCMPTC	58
4.6.3	LCMGLC	59
4.6.4	LCMPLC	59
4.7	Using variable-length character arrays	60
4.7.1	LCMGCD	60
4.7.2	LCMPCD	61
4.7.3	LCMGCL	62
4.7.4	LCMPCL	62
4.8	Dynamic allocation of an elementary blocks of data in ANSI C	63
4.8.1	LCMARA	63
4.8.2	LCMDRD	63
4.9	Abnormal termination of the execution	64
4.9.1	XABORT	64
5	The ISO Fortran CLE-2000 API	65
5.1	Management of Fortran files outside CLE-2000	65
5.1.1	KDROPN	65
5.1.2	KDRCLS	65
5.2	Management of word-addressable (KDI) files outside CLE-2000	66
5.2.1	KDIOP	66
5.2.2	KDIGET	66
5.2.3	KDIPUT	66
5.2.4	KDICL	67
5.3	Management of Fortran and KDI files used as CLE-2000 parameters	67
5.3.1	FILOPN	67
5.3.2	FILCLS	67
5.3.3	FILUNIT	68
5.3.4	FILKDI	68
5.4	The main entry point for CLE-2000	68
5.4.1	KERNEL	68
5.4.2	DUMMOD	69
5.5	The free-format input reader	72
5.5.1	REDOPN	72
5.5.2	REDGET	72
5.5.3	REDPUT	73
5.5.4	REDCLS	73
References		75
Index		76

1 The GANLIB Version 5 architecture

The GANLIB is a small library that is linked to a software application in order to facilitate modularity, interoperability, and to bring generic capabilities in term of data transfer. The GANLIB is an application programming interface (API) made of subroutines that are called by the software application (e.g., a lattice code) or by the multi-physics surrounding application. In other words, the GANLIB acts as a standardized interface between the software application and the multi-physics application, as depicted in Figure 1.

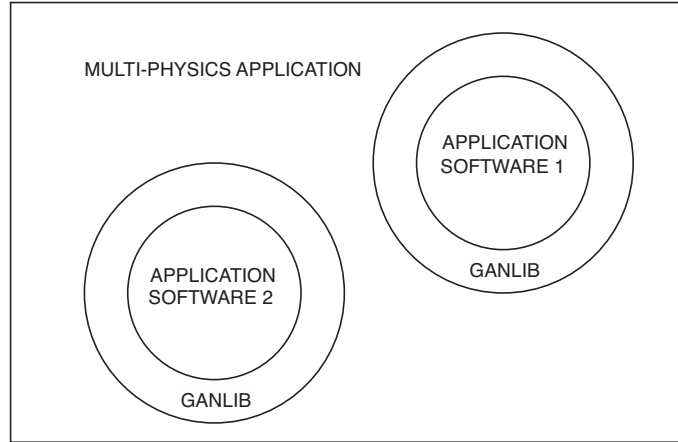


Figure 1: Implementing a multi-physics application.

The GANLIB is made of two distinct and inter-related components:

- CLE-2000 is a compact supervisor responsible for the free-format recovery of input data, for the modularization of the software application and for the insertion of loops and control statements in the input data flow. CLE-2000 permits the conception of *computational schemes*, dedicated to specific engineering studies, without any need for recompilation of the software application.^[1]
- LCM objects are data structures used to transfer data between modules of the software application and towards the multi-physics application. LCM objects are structures made of *associative table* and *heterogeneous lists*. These structures are either *memory resident* or *persistent* (i.e., stored in a file). The LCM object API is implemented with access efficiency as its first requirement, even for frequent calls with small chunks of data.^[2]

The GANLIB Version 5 is implemented in the ANSI C programming language^[3], in order to maximize its compatibility in a multi-physics environment where different components are implemented in various programming languages (C++, Fortran, Java, etc.). The GANLIB Version 5 is *64-bit clean*, another benefit of using an ANSI C implementation. This last property allows the execution of software applications with 32-bit integers and 64-bit addresses. Specific Fortran APIs are also available and are implemented according to the C interoperability mechanism, available in Fortran 2003 and standardized by the International Organization for Standardization (ISO). This architecture is 64-bit clean.^[4]

1.1 From Versions 3 or 4 to Version 5

The Version 3 and Version 4 software applications available at GAN are using a legacy GANLIB, implemented in FORTRAN 77, and relatively unchanged for more than 15 years.^[5] The only addition in Version 4 are the heterogeneous lists within LCM objects. This Fortran implementation is *not* ISO standard and *not* 64-bit clean. However, the corresponding API is mature and efficient, two qualities that we want to preserve.

A software application is not 64-bit clean when 32-bit integers are used to store addresses (or differences between two addresses). This nasty operation is possible in ANSI C but can always be avoided. Unfortunately, this operation is extensively used in software applications DRAGON, TRIVAC and DON-JON Versions 3 or 4, due to design constraints related to the choice of FORTRAN 77 as programming language.

Versions 3 or 4 software applications can be re-implemented around the Version 5 GANLIB in order to become ISO standard and 64-bit clean. However, the conversion process is not automatic and is time-consuming. Two major modifications must be done:

1. All variables containing addresses of LCM objects must be declared as `TYPE(C_PTR)` instead of been declared as `INTEGER`. The intrinsic type `TYPE(C_PTR)` is available in Fortran 2003, as defined by ISO.
2. Every call to the `SETARA` subroutine of the GANLIB must be replaced by an `ALLOCATE` statement and every call to the `RLSARA` subroutine must be replaced by a `DEALLOCATE` statement. Statements `ALLOCATE` and `DEALLOCATE` are available in Fortran 90, as defined by ISO. The `ALLOCATABLE` attribute is used to identify allocated arrays. Blank common are no longer required as reference addresses. This modification is the more time-consuming of the two.

Implementing software applications in Fortran 2003 offers the opportunity to use advanced features of this language, such as pointers, Fortran modules (not to be confused with CLE-2000 modules) and polymorphism. However, this is a programming style issue which is independent of the selection of GANLIB Version 5. It is possible, as a pragmatic choice, to keep the Fortran-77 programming style and just use the GANLIB Version 5, `TYPE(C_PTR)` types and `ALLOCATABLE` arrays.

2 The ANSI C CLE-2000 API

2.1 The main entry point for CLE-2000

The CLE-2000 supervisor have been entirely reprogrammed in ANSI C in its GANLIB Version 5 implementation. Its main entry point is function `cle2000_c()` that can be used to execute a *CLE-2000 source file* which can be a *main procedure* (a sequential ASCII file with `.x2m` suffix) or a *parametrized procedure* (a sequential ASCII file with `.c2m` suffix). Parametrized procedures can be called by function `cle2000_c()` or by other CLE-2000 procedures. Function `cle2000_c()` is therefore recursive. A computational scheme is a set of parametrized procedures.

2.1.1 `cle2000_c`

The general specification of function `cle2000_c()` is

```
cle2000_c(ilevel, dummod, filenm, iprint, my_param);
```

input parameters:		
ilevel	<i>int_32</i>	recursivity level of <code>cle2000_c()</code> call. We recommend to call <code>cle2000_c()</code> from the main entry point with <code>ilevel = 1</code> .
dummod	<i>int_32 (*)()</i>	external ANSI C function (or C-interoperable Fortran-2003 function) responsible for dispatching the execution among calculation modules. Note that the calculation modules can be implemented in any language that is interoperable with ANSI C.
filenm	<i>char*</i>	name of sequential ASCII file containing the CLE-2000 source file, without the <code>.c2m</code> suffix. Can be set to " " (corresponding to <code>stdin</code> in ANSI C, or unit 5 in Fortran). The name is null terminated.
iprint	<i>int_32</i>	print parameter (set to zero for no print).
my_param	<i>lifo*</i>	last-in-first-out (lifo) stack containing LCM (or XSM) objects, files and/or CLE-2000 variables that are exchanged with the CLE-2000 procedure. Set <code>my_param = NULL</code> if no information is exchanged. The specification of <code>my_param</code> is detailed in Sect. 2.3 .

value of the function:	
<i>int_32</i>	error code equal to zero if the execution of the CLE-2000 source file is successful. Equal to the error code otherwise.

2.1.2 `dummod`

Function `dummod()` is an external ANSI C function (or C-interoperable Fortran-2003 function) responsible for dispatching the execution among calculation modules. A specific version, named `ganmod()`, is used to dispatch the execution among the modules of the GANLIB. Its specifications are:

```
dummod(cmodul, nentry, hentry, ientry, jentry, kentry, hparam);
```

input parameters:		
cmodul	<i>char*</i>	name of the calculation module to execute
nentry	<i>int_32</i>	number of parameters (LCM objects or files) for this call
hentry	<i>char (*)[13]</i>	names of the parameters as known in the CLE-2000 procedure
ientry	<i>int_32*</i>	types of the parameters. = 1: memory-resident LCM object; = 2: persistent LCM object (stored in a XSM file); = 3: sequential binary file; = 4: sequential ascii file; = 5: direct access file.
jentry	<i>int_32*</i>	access mode of the parameters. = 0: the object is created; = 1: the object is opened for modifications ; = 2: the object is opened in read-only mode.
kentry	<i>lcm**</i>	equal to the address of the LCM object corresponding to a parameter or set to NULL if the parameter is a file
hparam	<i>char (*)[73]</i>	names of the parameters as known by the operating system

value of the function:	
<i>int_32</i>	error code equal to zero if the execution of dummod() is successful. Equal to the error code otherwise.

2.1.3 Calling a main CLE-2000 procedure

The simplest situation occurs when a main CLE-2000 procedure is called. This situation corresponds to the case where an application software is run in stand-alone mode. In this case, it is sufficient to write a main program calling a main CLE-2000 procedure. The main program can be written in ANSI C (as in the following example) or as a C-interoperable Fortran-2003 program. A main CLE-2000 procedure has no in-out CLE-2000 variables and no in-out parameters.

In the following example, an application software contains three modules, named MOD1:, MOD2 and MOD3, respectively. A main program is simply written as

```
#include <string.h>
#include "cle2000.h"
main()
{
    int_32 iprint = 0;
    int_32 ier, ilevel = 1;
    int_32 ganmod();

    ier = cle2000_c(ilevel, &ganmod, " ", iprint, NULL);
    printf("end of execution; ier=%d\n", ier);
}
```

The **ganmod()** function is another developer-supplied function that is responsible for dispatching the execution among modules MOD1:, MOD2 or MOD3. The **ganmod()** function is responsible for opening any file that can be requested by these modules. This open/close operation may be different, depending if the modules are programmed in ANSI C (as in this example) or in another language.

```
#include <string.h>
#include <stdio.h>
#include "cle2000.h"
int_32 ganmod(char *cmodul, int_32 nentry, char (*hentry)[13], int_32 *ientry,
    int_32 *jentry, lcm **kentry, char (*hparam)[73])
```



```

{
    int_32 iloop1, ier;
    FILE *kentry_file[maxent];
    char hsmg[132];

/* open files */
    for (iloop1 = 0; iloop1 < nentry; ++iloop1) {
        if (ientry[iloop1] >= 3) {
            char *mode;
            if ((ientry[iloop1] == 3) && (jentry[iloop1] == 0)) {
                strcpy(mode, "w");
            } else if ((ientry[iloop1] == 3) && (jentry[iloop1] == 1)) {
                strcpy(mode, "a");
            } else if ((ientry[iloop1] == 3) && (jentry[iloop1] == 2)) {
                strcpy(mode, "r");
            } else if ((ientry[iloop1] == 4) && (jentry[iloop1] == 0)) {
                strcpy(mode, "wb");
            } else if ((ientry[iloop1] == 4) && (jentry[iloop1] == 1)) {
                strcpy(mode, "ab");
            } else if ((ientry[iloop1] == 4) && (jentry[iloop1] == 2)) {
                strcpy(mode, "rb");
            } else {
                sprintf(hsmg, "ganmod: type not supported for file %s", hentry[iloop1]);
                xabort_c(hsmg);
            }
            kentry_file[iloop1] = fopen(hparam[iloop1], mode);
            if (kentry_file[iloop1] == NULL) {
                sprintf(hsmg, "ganmod: unable to open file %s", hentry[iloop1]);
                xabort_c(hsmg);
            }
        } else {
            kentry_file[iloop1] = NULL;
        }
    }

/* call modules */
    if(strcmp(cmodul, "MOD1:") == 0) {
        mod1(nentry, hentry, ientry, jentry, kentry, kentry_file);
    } else if(strcmp(cmodul, "MOD2:") == 0) {
        mod2(nentry, hentry, ientry, jentry, kentry, kentry_file);
    } else if(strcmp(cmodul, "MOD3:") == 0) {
        mod3(nentry, hentry, ientry, jentry, kentry, kentry_file);
    } else {
        return 1;
    }

/* close files */
    for (iloop1 = 0; iloop1 < nentry; ++iloop1) {
        if (ientry[iloop1] >= 3) {
            ier = fclose(kentry_file[iloop1]);
            if (ier != 0) {
                sprintf(hsmg, "ganmod: unable to close file %s", hentry[iloop1]);
                xabort_c(hsmg);
            }
        }
    }
}

```

```

        }
    }
}
return 0;
}

```

2.1.4 Calling a parametrized CLE-2000 procedure

In cases where an application software is called from a multi-physics application, it is likely that the multi-physics application will need to call parametrized CLE-2000 procedures (with “.c2m” suffix). This approach provides an efficient way of communication between the application software and the multi-physics application. It also permit to develop computational schemes outside the scope (i.e., independently) of the multi-physics application. Parameters are either LCM objects (memory-resident) or files that are managed by the operating system. Multi-physics applications are generally programmed in C++ or in Java. In the latter case, *Java Native Interfaces* (JNIs) are required to allow this communication.

In the following example, a parametrized procedure, **TESTproc.c2m**, take two object parameters and three CLE-2000 input variables. Note that the CLE-2000 variables are always defined after LCM and file objects. The first parameter, **MACRO_ASCII**, is an ASCII file written by the procedure and containing an export of the information pointed by the second parameter **MACRO**. This second parameter is a memory resident LCM object containing a Macrolib. It is accessed in **read-only** mode. The procedure also prints a table-of-content of the root directory of **MACRO**, using the **UTL:** module of the GANLIB. The procedure **TESTproc.c2m** is implemented as

```

REAL KEFF1 KEFF2 ;
INTEGER I123 ;
PARAMETER MACRO_ASCII MACRO ::
    EDIT 1
    ::: SEQ_ASCII MACRO_ASCII ;
    ::: LINKED_LIST MACRO ;
;
:: >>KEFF1<< >>KEFF2<< >>I123<< ;
MODULE UTL: END: ;
*
UTL: MACRO :: DIR ;
MACRO_ASCII := MACRO ;
ECHO "KEFF1=" KEFF1 ""KEFF2=" KEFF2 "I123=" I123 ;
ECHO "procedure TESTproc completed" ;
END: ;
QUIT "XREF" .

```

More information about the development of CLE-2000 procedures can be found in Ref. [1](#).

The next ANSI C function is an example of how a multi-physics application can call such a procedure. A LCM object containing a Macrolib is first created by importing its information from an existing ASCII file named **Macrolib**. Next, a call to function **cle2000_c()** is performed to execute **TESTproc.c2m**. The corresponding **main** program is written

```

#include <string.h>
#include <stdlib.h>
#include "cle2000.h"
main()
{
    int_32 iprint = 0;

```

```

    int_32 ier, ilevel = 1;
    FILE *filein;
    char cproce[13];
    int_32 ganmod();
    lcm *my_lcm;
    lifo *my_lifo;
    lifo_node *my_node;

/* create the LCM object containing a Macrolib */
    filein = fopen("Macrolib", "r");
    lcmop_c(&my_lcm, "MACRO1", 0, 1, iprint);
    lcmexp_c(&my_lcm, iprint, filein, 2, 2);
    fclose(filein);
    lcmllib_c(&my_lcm);
    lcmcl_c(&my_lcm, 1);

/* construct the lifo stack */
    cleopn(&my_lifo);
    /* node 1 */
    my_node = (lifo_node *) malloc(sizeof(lifo_node));
    strcpy(my_node->name, "MACRO_ASCII1"); strcpy(my_node->OSname, "my_ascii_file");
    my_node->type = -6;
    clepush(&my_lifo, my_node);
    /* node 2 */
    my_node = (lifo_node *) malloc(sizeof(lifo_node));
    strcpy(my_node->name, "MACRO1"); strcpy(my_node->OSname, "MACRO1"); my_node->type = 3;
    my_node->value.lcm = my_lcm;
    clepush(&my_lifo, my_node);
    /* node 3 */
    my_node = (lifo_node *) malloc(sizeof(lifo_node));
    strcpy(my_node->name, "value1"); my_node->type = 12; my_node->value.fval = 1.703945;
    clepush(&my_lifo, my_node);
    /* node 4 */
    my_node = (lifo_node *) malloc(sizeof(lifo_node));
    strcpy(my_node->name, "value2"); my_node->type = 12; my_node->value.fval = 1.562276;
    clepush(&my_lifo, my_node);
    /* node 5 */
    my_node = (lifo_node *) malloc(sizeof(lifo_node));
    strcpy(my_node->name, "value3"); my_node->type = 11; my_node->value.ival = 12345;
    clepush(&my_lifo, my_node);

/* call the parametrized procedure */
    strcpy(cproce, "TESTproc");
    ier = cle2000_c(ilevel, &ganmod, cproce, iprint, my_lifo);
    if (ier != 0) xabort_c("example2.1.5: cle2000 failure");

/* erase the lifo stack */
    while (my_lifo->nup > 0) {
        my_node = clepop(&my_lifo);
        free(my_node);
    }
    clecls(&my_lifo);
    printf("successful end of execution\n");

```

```
}

```

2.1.5 Calling a CLE-2000 procedure with in-out CLE-2000 variables

The CLE-2000 API also offers the possibility to exchange CLE-2000 variables with a procedure. The following CLE-2000 procedure permits to compute the factorial of a number, as proposed in Ref. 1. Here, `n` and `n_fact` are input and output CLE-2000 variable, respectively. The `fact.c2m` procedure is written

```
!
! Example of a recursive procedure.
!
! input to "fact": *n*
! output from "fact": *n_fact*
!
INTEGER    n n_fact prev_fact ;
:: >>n<< ;
IF n 1 = THEN
    EVALUATE n_fact := 1 ;
ELSE
    EVALUATE n := n 1 - ;
    ! Here, "fact" calls itself
    PROCEDURE fact ;
    fact :: <<n>> >>prev_fact<< ;
    EVALUATE n_fact := n 1 + prev_fact * ;
ENDIF ;
:: <<n_fact>> ;
QUIT " Recursive procedure *fact* XREF " .

```

This procedure can be called from a program implemented in ANSI C, using

```
#include <string.h>
#include <stdlib.h>
#include "cle2000.h"
main()
{
    int_32 iprint = 0;
    int_32 ier, ilevel = 1;
    char cproce[13];
    int_32 ganmod();
    lifo *my_lifo;
    lifo_node *my_node;

    /* construct the lifo stack */
    cleopn(&my_lifo);
    /* node 1 */
    my_node = (lifo_node *) malloc(sizeof(lifo_node));
    strcpy(my_node->name, "input_val"); my_node->type = 11; my_node->value.ival = 5;
    clepush(&my_lifo, my_node);
    /* node 2 */
    my_node = (lifo_node *) malloc(sizeof(lifo_node));
    strcpy(my_node->name, "output_val"); my_node->type = -11;
    clepush(&my_lifo, my_node);

```

```

/* call the procedure with in-out CLE-2000 variables*/
strcpy(cproce, "fact");
ier = cle2000_c(ilevel, &ganmod, cproce, iprint, my_lifo);
if (ier != 0) xabort_c("fact: cle2000 failure");

/* recover and erase the lifo stack */
printf("\noutput stack:\n");
while (my_lifo->nup > 0) {
    my_node = clepop(&my_lifo);
    printf("node %d (name=%12s) ---> %d\n", my_lifo->nup, my_node->name,
        my_node->value.ival);
    free(my_node);
}
clecls(&my_lifo);
printf("successful end of execution\n");
}

```

2.2 Calling a calculation module without a CLE-2000 procedure

The GANLIB API also provides the possibility to call directly a calculation module without a CLE-2000 procedure. This capability is required in the first-generation Jargon framework, as presented in Ref. 6. The actual implementation does not support CLE-2000 variables. A calculation module with “>> <<” variables must therefore be encapsulated in a CLE-2000 procedure.

2.2.1 *clemod_c*

The general specification of function `clemod_c()` is

```
clemod_c(cmodul, filein, nentry, hentry, ientry, jentry, kentry, hparam, dummod);
```

input parameters:		
cmodul	<i>char*</i>	name of the calculation module to execute
filein	<i>FILE*</i>	sequential ASCII file containing the data for module cmodul (i.e., the data between the “:” and the “;”). Can be set to stdin (standard input, or unit 5 in Fortran)
nentry	<i>int_32</i>	number of parameters (LCM objects or files) that are exchanged with the CLE-2000 procedure. nentry = 0 if no parameters are exchanged.
hentry	<i>char (*)[13]</i>	names of these parameters, as known by the calculation module. Each name is a character string with a maximum of 12 characters.
ientry	<i>int_32*</i>	types of each parameter. = 1: memory-resident LCM object; = 2: persistent LCM object (stored in a XSM file); = 3: sequential binary file; = 4: sequential ascii file; = 5: direct access file.
jentry	<i>int_32*</i>	mode of each parameter. = 0: the object is created; = 1: the object is opened for modifications ; = 2: the object is opened in read-only mode.
kentry	<i>lcm**</i>	addresses of the lcm objects (for parameters that are LCM objects). Set to NULL for parameters that are files.
hparam	<i>char (*)[73]</i>	names of these parameters, as known by the operating system. Each name is a character string with a maximum of 72 characters.
dummod	<i>int_32 (*)()</i>	external ANSI C function (or C-interoperable Fortran-2003 function) responsible for dispatching the execution among calculation modules. Note that the calculation modules can be implemented in any language that is interoperable with ANSI C.

value of the function:	
<i>int_32</i>	error code equal to zero if the execution of the calculation module is successful. Equal to the error code otherwise.

In the following example, function `clmod.c()` is used to call a calculation module of the GANLIB. A LCM object containing a Macrolib is first created by importing its information from an existing ASCII file named `Macrolib`. Module `UTL:` is called with this `read-only` Macrolib as unique parameter:

```
#include <string.h>
#include "cle2000.h"
#define maxent 64 /* maximum number of module arguments */
main()
{
    int_32 ganmod();
    char hentry[maxent][13], hparam[maxent][73];
    int_32 ier, nentry, ientry[maxent], jentry[maxent];
    lcm * my_lcm, *kentry[maxent];
    FILE *filein;

    /* create the LCM object containing a Macrolib */
    filein = fopen("Macrolib", "r");
    lcmop_c(&my_lcm, "MACRO", 0, 1, 99);
    lcmexp_c(&my_lcm, 99, filein, 2, 2);
    fclose(filein);

    /* create a file containing the UTL: data */
    filein = fopen("UTLdata", "r");

    /* construct the parameter */
```

```

    nentry = 1 ;
    strcpy(hentry[0], "MACRO"); strcpy(hparam[0], "MACRO"); ientry[0]=1; jentry[0]=2;
    kentry[0]=my_lcm;

/* execute the module */
    ier = clemod_c("UTL:", filein, nentry, hentry, ientry, jentry, kentry, hparam,
        &ganmod);
    lcmcl_c(&my_lcm, 1);
    fclose(filein);
    printf("end of execution; ier=%d\n", ier);
}

```

The ASCII file UTLdata contains the data for module UTL:. Here, it is defined as

```

DIR STEP UP GROUP
STEP AT 1 DIR STEP DOWN
STEP DOWN ;

```

2.3 Management of the last-in-first-out (lifo) stack

A last-in-first-out (lifo) stack manage the stored data so that the last data stored in the stack is the first data removed from the stack. This means that a POP function retrieves the values most recently stored with a PUSH function. CLE-2000 uses one lifo stack to manage information used within each specific CLE-2000 procedure instance and one lifo stack as dummy parameter list each time a CLE-2000 procedure is called.

In case where a CLE-2000 procedure is called from a multi-physics environment, the parameter information is first integrated in a lifo stack *before* calling function `cle2000_c()`. After execution of the procedure, output parameter information is recovered from the lifo stack. The lifo stack can contain LCM (or XSM) objects, files and/or CLE-2000 variables. The lifo stack is constructed as a linked list of nodes, each node containing a single parameter. Three important rules must be satisfied:

- LCM (or XSM) objects and files must be defined prior to CLE-2000 variables in the lifo stack used as parameter information.
- LCM (or XSM) objects and files must be closed when included in the lifo stack.
- Output nodes are also included in the lifo stack before calling function `cle2000_c()`, but *with negative type* component and *without value* component.

The specification of a lifo node is:

```

typedef struct LIFO_NODE {          /* node in last-in-first-out (lifo) stack */
    int_32 type;                    /* type of node: 3= lcm object; 4= xsm file; 5= seq binary;
                                   6= seq ascii; 7= da binary; 11= integer value; 12= real value;
                                   13= character string; 14= double precision value;
                                   15= logical value */
    int_32 access;                  /* 0=creation mode/1=modification mode/2=read-only mode */
    int_32 lparam;                  /* record length for DA file objects */
    union {
        int_32 ival;               /* integer or logical value */
        float_32 fval;             /* real value */
        double dval;              /* double precision value */
        lcm *lcm;                  /* handle towards a LCM object */
        char hval[73];             /* character value */
    } value;
}

```

```

    struct LIFO_NODE *daughter; /* address of the daughter node in stack */
    char name[13];             /* name of node in the calling script */
    char name_daughter[13];    /* name of node in the daughter script */
    char OSname[73];           /* physical filename */
} lifo_node ;

```

life_node components:		
type	int_32	type of data in node. = ± 3 : LCM object; = ± 4 : XSM file; = ± 5 : sequential binary file; = ± 6 : sequential ascii file; = ± 7 : direct access binary file; = ± 11 : integer CLE-2000 value; = ± 12 : real CLE-2000 value; = ± 13 : character string (null-terminated); = ± 14 : double precision CLE-2000 value; = ± 15 : logical CLE-2000 value. A positive value indicates that an input value is provided; a negative value indicates that <i>no</i> input value is provided so that the node is <i>empty</i> . Empty nodes are defined to receive calculational results.
access	int_32	access state of data in node. = 0: creation mode; = 1: modification mode; = 2: read-only mode. This information is used internally in cle2000_c() function.
lparam	int_32	record length (in bytes) for DA file objects. This data is given if and only if type = 7.
value.ival	int_32	integer or logical CLE-2000 value. This data is given or is available at output if and only if type = 11 or = 15.
value.fval	float_32	real CLE-2000 value. This data is given or is available at output if and only if type = 12.
value.hval	char[73]	character string CLE-2000 value. This data is given or is available at output if and only if type = 13.
value.dval	double	double precision CLE-2000 value. This data is given or is available at output if and only if type = 14.
value.lcm	lcm*	LCM object (memory-resident). This data is given or is available at output if and only if type = 3. The LCM object is closed.
daughter	lifo_node*	address of the daughter node in stack. This information is used by the lifo utility to construct the linked list of nodes.
name	char[13]	name of node in the calling script.
name_daughter	char[13]	name of node in the daughter script. This name is used internally in cle2000_c() function.
OSname	char[73]	name of node as known by the operating system. In the case of a LCM object, it is the name given to lcmop_c() function. In the case of a file, it is the operating system name of the file. The LCM object or file is closed. This data is given if and only if type \leq 10.

The following functions are used to manage the lifo stack.

2.3.1 cleopn

Create an empty lifo stack.

```
cleopn(my_lifo);
```


output parameter:		
my_lifo	<i>lifo**</i>	address of the empty lifo stack.
value of the function:		
<i>void</i>		

2.3.2 clepop

Remove the “last-in” node from the lifo stack.

```
clepop(my_lifo);
```

input parameter:		
my_lifo	<i>lifo**</i>	address of the lifo stack.
value of the function:		
<i>lifo_node*</i>		node removed from the lifo stack

2.3.3 clepush

Remove the “last-in” node from the lifo stack.

```
clepush(my_lifo, my_node);
```

input parameters:		
my_lifo	<i>lifo**</i>	address of the lifo stack.
my_node	<i>lifo_node*</i>	node to add to the lifo stack.
value of the function:		
<i>void</i>		

2.3.4 clecls

Delete an empty lifo stack.

```
clecls(my_lifo);
```

input parameter:		
my_lifo	<i>lifo**</i>	address of the empty lifo stack.
value of the function:		
<i>int_32</i>		error code. = 0: successful; = -1: the lifo stack is not empty.

2.3.5 clenode

Return the node with name **my_name**. The lifo stack is not modified.

```
clenode(my_lifo, my_name);
```

input parameters:		
my_lifo	<i>lifo**</i>	address of the lifo stack.
my_name	<i>char*</i>	name of the node. The name is null-terminated.
value of the function:		
<i>lifo_node*</i>		node of name my_name or NULL if the node doesn't exist.

2.3.6 clepos

Return the **ipos**-th node in the stack. The lifo stack is not modified.

```
clepos(my_lifo, ipos);
```

input parameters:		
my_lifo	<i>lifo**</i>	address of the lifo stack.
ipos	<i>int_32</i>	position of the node in the stack.
value of the function:		
<i>lifo_node*</i>		ipos -th node or NULL if the node doesn't exist.

2.3.7 clelib

Print a table-of-content for the lifo stack.

```
clelib(my_lifo);
```

input parameter:		
my_lifo	<i>lifo**</i>	address of the lifo stack.
value of the function:		
<i>void</i>		

2.4 The free-format input reader

The free-format input reader of CLE-2000 is implemented using four functions: **redopn_c()**, **redget_c()**, **redput_c()** and **redcls_c()**. Only **redget_c()** and **redput_c()** are expected to be used in an application software.

2.4.1 redopn_c

Function **redopn_c()** is called to open the input reader. The general specification of function **redopn_c()** is

```
redopn_c(iinp1, iout1, hout1, nrec);
```

input parameters:		
iinp1	<i>kdi_file*</i>	KDI object containing the CLE-2000 input data, as computed by <code>clepil()</code> and <code>objpil()</code> functions of CLE-2000.
iout1	<i>FILE*</i>	sequential ASCII file used to write execution messages. Can be set to <code>stdout</code> .
hout1	<i>char*</i>	name of the sequential ASCII file used to write execution messages.
nrec	<i>int_32</i>	record index where reading occurs. Can be set to zero at first call. Set to the value returned by <code>redcls_c()</code> at subsequent calls.

value of the function:	
<i>void</i>	

2.4.2 *redget_c*

Function `redget_c()` is called within modules of the application software to recover the module-specific input data. The general specification of function `redget_c()` is

```
redget_c(ityp, nitma, flott, text, dflot);
```

output parameters:		
ityp	<i>int_32*</i>	type of the CLE-2000 variable. A negative value indicates that the variable is to be computed by the application software and returned towards CLE-2000 using a call to <code>redput_c</code> . = ± 1 : integer type; = ± 2 : real (single precision) type; = ± 3 : string type; = ± 4 : double precision type; = ± 5 : logical type.
nitma	<i>int_32*</i>	integer input value when <code>ityp</code> = 1 or = 5; number of characters when <code>ityp</code> = 3.
flott	<i>float_32*</i>	real input value when <code>ityp</code> = 2.
text	<i>char[73]</i>	character string input value when <code>ityp</code> = 3.
dflot	<i>double_64*</i>	double precision input value when <code>ityp</code> = 4.
value of the function:		
<i>void</i>		

2.4.3 *redput_c*

Function `redput_c()` is called within modules of the application software to make information computed by the module available as CLE-2000 variables to the CLE-2000 procedure. The application software must first call `redget_c()` and obtain a negative value of `ityp`. A call to `redput_c()` is next performed with its first parameter set to $-\text{ityp}$ (now, a positive value) and with the corresponding value of the parameter. The general specification of function `redput_c()` is

```
redput_c(ityp, nitma, flott, text, dflot);
```

input parameters:		
ityp	<i>int_32*</i>	type of the CLE-2000 variable. = 1: integer type; = 2: real (single precision) type; = 3: string type; = 4: double precision type; = 5: logical type.
nitma	<i>int_32*</i>	integer output value when ityp = 1 or = 5; number of characters when ityp = 3.
flott	<i>float_32*</i>	real output value when ityp = 2.
text	<i>char*</i>	character string output value when ityp = 3.
dflot	<i>double_64*</i>	double precision output value when ityp = 4.

value of the function:	
<i>void</i>	

2.4.4 redcls_c

Function `redcls_c()` is called to close the input reader. The general specification of function `redcls_c()` is

```
redcls_c(iinp1, iout1, hout1, nrec)
```

output parameters:		
iinp1	<i>kdi_file**</i>	KDI object containing the CLE-2000 input data.
iout1	<i>FILE**</i>	sequential ASCII file used to write execution messages.
hout1	<i>char[73]</i>	name of the sequential ASCII file used to write execution messages.
nrec	<i>int_32*</i>	record index where reading occurs.
value of the function:		
<i>void</i>		

2.5 Defining built-in constants in CLE-2000

CLE-2000 has pre-defined built-in constants, either with mathematical meaning (e.g., π) or with physical meaning. Currently, available physical constants are related to reactor physics. In future, one may want to include more physical constants. Here is the specification of the function available inside CLE-2000 to define these constants.

2.5.1 clecst

Function `dumcst()` is an external ANSI C function implementing pre-defined parametric constants. A standard version is available in the GANLIB with name `clecst()`. It is specified as

```
clecst(cparm, ityp, nitma, flott, text, dflot);
```

input parameter:		
cparm	<i>char*</i>	name of the parametric constant (name starting with \$)

output parameters:		
ityp	<i>int_32*</i>	type of the parametric constant ($1 \leq \text{ityp} \leq 5$)
nitma	<i>int_32*</i>	integer value of the parametric constant if ityp = 1; logical value (=1: true/=1: false) of the parametric constant if ityp = 5; number of characters in the string if ityp = 3.
flott	<i>float_32*</i>	real value of the parametric constant if ityp = 2
text	<i>char*</i>	character string value of the parametric constant if ityp = 3
dflot	<i>double_64*</i>	double precision value of the parametric constant if ityp = 4
value of the function:		
<i>int_32</i>		error code equal to zero if the execution of clecst() is successful. Equal to the error code otherwise.

3 The ANSI C LCM API

LCM objects are data structures, implemented in ANSI C, with characteristics of *associative tables* (a.k.a., dictionaries or hash tables) and/or *heterogeneous lists* (a.k.a., cell arrays). These data structures are either stored in memory or are persistent (i.e., stored in a file). These objects are primarily accessed via an API implemented in ANSI C. Access by other languages is possible via *specific bindings* that are also described in this report. Deep copy and serialization utilities are available.

Persistence is implemented using XSM data structures, together with another API implemented in ANSI C. xsm files are used in this case. However, the XSM API is invoked from within LCM and a developer using the GANLIB never has to call it directly.

The LCM API was implemented in such a way that

- the access from ANSI C or from Fortran is highly optimized, even for frequent calls with small chunks of data.
- the access from other languages (Matlab, Python, Java, or Objective C) permits a complete read/write access of the totality of information contained in the object.

This technical report contains the precise description of each ANSI C function available in the LCM API and dedicated to a programmer using the GANLIB Version 5.

A LCM object is a collection of the following elements:

Associative tables

An associative table is equivalent to a Python dictionary or to a Java hash table. Each element of an associative table is an association between a 12-character name and a block of data. A block of data can be an array of some elementary type, another associative table or an heterogeneous list. Tree structures can be constructed that way.

Heterogeneous list

An heterogeneous list is an ordered set of blocks of information (referred as “0”, “1”, “2”, etc.). A block of data can be an array of some elementary type, another associative table or an heterogeneous list.

Array of elementary type

An array of elementary type is a set of consecutive values in memory, all of the same type. The type is selected in the following table:

index	array of ...	type
1	32-bit integer	int_32
2	32-bit real	float_32
3	character*4 strings	
4	64-bit real	double_64
5	32-bit logical	int_32
6	64-bit complex	

Any ANSI C function calling the LCM API must use an include file of the form

```
#include "lcm.h"
```

Each LCM object has a *root associative table* from which the complete object is constructed.

3.1 General utility functions

3.1.1 *strcut.c*

Copy **n** characters from string **ct** to **s**. Eliminate leading ' ' and '\0' characters in **s**. Terminate **s** with a '\0'.

```
strcut_c(s, ct, n);
```

input parameters:		
ct	<i>char*</i>	character variable of length n . May not be null-terminated.
n	<i>int_32</i>	length of ct .

output parameter:		
s	<i>char*</i>	null terminated string.
value of the function:		
<i>void</i>		

3.1.2 *strfil.c*

Copy **n** characters from string **ct** to **s**. Eliminate '\0' characters and pack with ' '. Assume that **ct** is null-terminated.

```
strfil_c(s, ct, n);
```

input parameters:		
ct	<i>char*</i>	null-terminated character variable.
n	<i>int_32</i>	expected length of s .

output parameter:		
s	<i>char*</i>	character variable of length n (not null-terminated).
value of the function:		
<i>void</i>		

3.2 Opening, closing and validation of LCM objects

3.2.1 *lcmop_c*

Open an LCM object (either memory resident or persistent). Obtain the address of the LCM object if it is created. Note that CLE-2000 is responsible to perform the calls to `lcmop_c` for the LCM objects that are used as parameters of a CLE-2000 module. The use of `lcmop_c` is generally restricted to the use of temporary LCM objects created within a CLE-2000 module.

```
lcmop_c(iplist,namp,imp,medium,impx);
```

input parameters:		
iplist	<i>lcm**</i>	address of the LCM object if imp =1 or imp =2. iplist corresponds to the address of the root associative table.
namp	<i>char[13]</i>	name of the LCM object if imp =0.
imp	<i>int_32</i>	=0 to create a new LCM object ; =1 to modify an existing LCM object; =2 to access an existing LCM object in read-only mode.
medium	<i>int_32</i>	=1 to use a memory-resident LCM object; =2 to use an XSM file to store the LCM object.
impx	<i>int_32</i>	print parameter. Equal to zero to suppress all printings.

output parameters:		
iplist	<i>lcm**</i>	address of an LCM object if imp =0.
namp	<i>char*</i>	name of the LCM object if imp =1 or imp =2.
value of the function:		
<i>void</i>		

3.2.2 *lcmcl_c*

Close an LCM object (either memory resident or persistent). Note that CLE-2000 is responsible to perform the calls to `lcmcl_c` for the LCM objects that are used as parameters of a CLE-2000 module. The use of `lcmcl_c` is generally restricted to the use of temporary LCM objects created within a CLE-2000 module.

A LCM object can only be closed if **iplist** points towards its root directory.

```
lcmcl_c(iplist,iact);
```

input parameters:		
iplist	<i>lcm**</i>	address of the LCM object (address of the root directory of the LCM object).
iact	<i>int_32</i>	=1 close the LCM object without destroying it; =2 and destroying it

output parameters:		
iplist	<i>lcm**</i>	iplist=null indicates that the LCM object is closed and destroyed. A memory-resident LCM object keeps the same address during its complete existence. A persistent LCM object is associated to an XSM file and is represented by a different value of iplist each time it is reopened.
value of the function:		
<i>void</i>		

3.2.3 *lcmval_c*

Function to validate a single block of data in a LCM object or the totality of the LCM object, starting from the address of an associative table. This function has no effect if the object is persistent. The validation consists to verify the connections between the elements of the LCM object, to verify that each element of the object is defined and to check for possible memory corruptions. If an error is detected, the following message is issued:

LCMVAL_C: BLOCK xxx OF THE TABLE yyy HAS BEEN OVERWRITTEN.

This function is called as

```
lcmval_c(iplist,namp);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table or of the heterogeneous list.
namp	<i>char*</i>	name of the block to validate in the associative table. If namp=' ' , all the blocks in the associative table are verified in a recursive way.

value of the function:	
<i>void</i>	

3.3 Interrogation of LCM objects

The data structures in an LCM object are self-described. It is therefore possible to interrogate them in order to know their characteristics.

		type of interrogation	
		father structure	information block
father	associative table	lcminf_c lcmnxt_c	lcmlen_c
	heterogeneous list	lcminf_c	lcmlcl_c

3.3.1 *lcmlen_c*

Function used to recover the length and type of an information block stored in an associative table (either memory-resident or persistent). The length is the number of elements in a daughter heterogeneous list or the number of consecutive 32-bit memory words required to store an array of elementary type. As an example, the length required to store an array of 64-bit real numbers is twice its dimension.

```
lcmilen_c(iplist,namp,ilong,itylcm);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the block.
output parameters:		
ilong	<i>int_32*</i>	length of the block in 32-bit words. =-1 for a daughter associative table; = <i>N</i> for a daughter heterogeneous list containing <i>N</i> components; =0 if the block doesn't exist.
itylcm	<i>int_32*</i>	type of information. =0 associative table; =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =10 heterogeneous list; =99 undefined (99 is returned if the block doesn't exist).
value of the function:		
<i>void</i>		

3.3.2 lcminf.c

Function used to recover general information about a LCM object.

```
lcminf_c(iplist,namlcm,nammy,empty,ilong,lcm,access);
```

input parameter:		
iplist	<i>lcm**</i>	address of the associative table or of the heterogeneous list.
output parameters:		
namlcm	<i>char*</i>	name of the LCM object.
nammy	<i>char*</i>	name of the associative table at address iplist . = '/' if the associative table is the root of the LCM object; = ' ' if the associative table is an heterogeneous list component.
empty	<i>int_32*</i>	32-bit integer variable set to 1 if the associative table is empty or set to 0 otherwise.
ilong	<i>int_32*</i>	= -1: iplist is an associative table; > 0: number of components in the heterogeneous list iplist .
lcm	<i>int_32*</i>	32-bit integer variable set to 1 if information is memory-resident or set to 0 if information is persistent (stored in an XSM file).
access	<i>int_32*</i>	32-bit integer variable set to the access mode of object. = 0: the object is closed (only available for memory-resident LCM objects); = 1: the object is open for modification; = 2: the object is open in read-only mode.
value of the function:		
<i>void</i>		

3.3.3 *lcmnxt_c*

Function used to find the name of the next block of data in an associative table. Use of *lcmnxt_c* is forbidden if the associative table is empty. The order of names is arbitrary. The search cycle indefinitely.

```
lcmnxt_c(iplist,namp);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of an existing block. namp=' ' can be used to obtain a first name to initiate the search.

output parameters:		
namp	<i>char*</i>	name of the next block. A call to <i>xabort_c</i> is performed if the associative table is empty.
value of the function:		
<i>void</i>		

3.3.4 *lcmlel_c*

Function used to recover the length and type of an information block stored in an heterogeneous list (either memory-resident or persistent). The length is the number of elements in a daughter heterogeneous list or the number of consecutive 32-bit memory words required to store an array of elementary type. As an example, the length required to store an array of 64-bit real numbers is twice its dimension.

```
lcmlel_c(iplist,iset,ilong,itylcm);
```

input parameters:		
iplist	<i>lcm**</i>	address of the heterogeneous list.
iset	<i>int_32</i>	index of the block in the list. The first element of the list is located at index 0.

output parameters:		
ilong	<i>int_32*</i>	length of the block in 32-bit words. =0 if the block doesn't exist.
itylcm	<i>int_32*</i>	type of information. =0 associative table; =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =10 heterogeneous list; =99 undefined (99 is returned if the block doesn't exist).
value of the function:		
<i>void</i>		

3.4 Management of the array of elementary type

Management of the array of elementary type can be performed *with* copy of the data (*lcmput_c*, *lcmget_c*, *lcmpdl_c* or *lcmgdl_c*) or *without* copy (*lcmppd_c*, *lcmgpd_c*, *lcmppl_c* or *lcmgpl_c*).

		type of operation	
		put	get
father	associative table	lcmput_c	lcmget_c
		lcmppd_c	lcmgpd_c
	heterogeneous list	lcmpdl_c	lcmgdl_c
		lcmppl_c	lcmgpl_c

3.4.1 lcmget_c

Function used to recover an information block (array of elementary type) from an associative table and to copy this data into memory.

```
lcmget_c(iplist,namp,data);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the block to recover. A call to xabort_c is performed if the block does't exist.

output parameters:		
data	<i>int_32*</i>	array of dimension \geq ilong in which the block is copied.
value of the function:		
<i>void</i>		

Function **lcmget_c** can be used to recover information of type other than *int_32** by using a cast operation. Here is an example:

```
#include "lcm.h"
...
float_32 data[5];
lcm *iplist;
iplist=... ;
lcmget_c(&iplist,namp,(int_32*)data);
```

Function **lcmget_c** can also be used to recover character-string information available in a block of the LCM object. It is also possible to use function **lcmgcd_c** presented in Section 3.7.1. In the following example, a block is stored in an associative table located at address **iplist**. The block has a name **namp** and a length equivalent to 5 32-bit words. The information is recovered into the integer array **idata** and transformed into a null-terminated character string **hname** using the **strcut_c** utility:

```
#include "lcm.h"
...
char *namp="...", hname[21];
int_32 idata[5];
lcm *iplist;
iplist=... ;
lcmget_c(&iplist,namp,idata);
strcut_c(hname,(char *)idata,20);
```

3.4.2 *lcmput_c*

Function used to store a block of data (array of elementary type) into an associative table. The information is copied from memory towards the LCM object. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed into a LCM object open in **read-only** mode.

```
lcmput_c(iplist,namp,ilong,itylcm,data);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the block.
ilong	<i>int_32</i>	length of the block in 32-bit words. If the array contains N real.64 values, ilong must be set to $2 \times N$.
itylcm	<i>int_32</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
data	<i>int_32*</i>	array of dimension \geq ilong to be copied into the LCM object. Array elements data[0] to data[ilong-1] must be initialized before the call to lcmput_c .

value of the function:	
<i>void</i>	

Function **lcmput_c** can be used to store information of type other than *int_32** by using a cast operation. Here is an example:

```
#include "lcm.h"
...
float_32 data[5];
lcm *iplist;
int_32 i;
iplist=... ;
for (i=0;i<5;i++) {
    data[i]=... ;
}
lcmput_c(&iplist,namp,5,2,(int_32*)data);
```

Function **lcmput_c** can also be used to store character-string information in an associative table of a LCM object. It is also possible to use function **lcmgcd_c** presented in Section 3.7.2. In the following example, a character string **hname** is first transformed into an integer array **idata** using the **strfil_c** utility. This array (block of data) is stored into the LCM object located at address **iplist**, using **lcmput_c**. The block has a name **namp**, a length equivalent to 5 32-bit words, and a type equal to 3.

```
#include "lcm.h"
...
char *namp="...", hname[20];
int_32 idata[5], il=5, it=3;
lcm *iplist;
iplist=... ;
strfil_c((char *)idata,hname,20);
lcmput_c(&iplist,namp,il,it,idata);
```

3.4.3 *lcmgpd.c*

Function used to recover the memory address of an information block (array of elementary type) from an associative table, *without* making a copy of the information. Use of this function must respect the following rules:

- If the information is modified after the call to `lcmgpd.c`, a call to `lcmppd.c` must be performed to acknowledge the modification.
- The block `*iofset` should never be released using a deallocation function such as `rlsara.c`, `free`, etc.
- The address `iofset` must never be copied into another variable.

Non respect of these rules may cause execution failure (core dump, segmentation fault, etc) without possibility to throw an exception.

A call to `lcmgpd.c` doesn't cause any modification to the LCM object. The data array information is accessed directly from memory locations `*iofset[0]` to `*iofset[ilong-1]` where `iofset` is the address returned by function `lcmgpd.c`.

```
lcmgpd_c(iplist,namp,iofset);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the block to recover. A call to <code>xabort.c</code> is performed if the block doesn't exist.

output parameters:		
iofset	<i>int_32**</i>	address of the data array.
value of the function:		
<i>void</i>		

3.4.4 *lcmppd.c*

Function used to store a block of data (array of elementary type) into an associative table *without* making a copy of the information. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed into a LCM object open in **read-only** mode.

If a block named namp already exists in the associative table, the address associated with namp is replaced by the new address and the information pointed by the old address is deallocated.

The array containing information stored by `lcmppd.c` must be originally allocated by a call of the form `iofset = setara_c(ilong)` or `iofset = (int_32*)malloc(ilong*sizeof(int_32))`.

```
lcmppd_c(iplist,namp,ilong,itylcm,iofset);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the block.
ilong	<i>int_32</i>	length of the block in 32-bit words.
itylcm	<i>int_32</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
iofset	<i>int_32*</i>	address of the data array, as returned by setara_c . Data elements iofset[0] to iofset[ilong-1] must be initialized before the call to lcmppd_c .

value of the function:	
<i>void</i>	

The information block of address **iofset** will automatically be deallocated using function **rlsara_c** at closing time of the LCM object. Situations exist where this block is shared with data structures other than LCM, and where the block must *not* be deallocated by the LCM API. In this case, it is imperative to follow the call to **lcmppd_c** by a call to function **refpush** of the form:

```
refpush(iplist,iofset);
```

3.4.5 *lcmdel_c*

Function used to erase an information block or a daughter heterogeneous list stored in a memory-resident associative table. Function **lcmdel_c** *cannot* be used with persistent LCM objects.

```
lcmdel_c(iplist,namp);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the block to erase.

value of the function:	
<i>void</i>	

3.4.6 *lcmgdl_c*

Function used to recover an information block (array of elementary type) from an heterogeneous list and to copy this data into memory.

```
lcmgdl_c(iplist,iset,data);
```

input parameters:		
iplist	<i>lcm**</i>	address of the heterogeneous list.
iset	<i>int_32</i>	index of the block in the heterogeneous list. A call to xabort_c is performed if the block doesn't exist. The first element of the list is located at index 0.

output parameters:		
data	<i>int_32*</i>	array of dimension \geq ilong in which the block is copied.
value of the function:		
<i>void</i>		

Function `lcmgdl_c` can be used to recover character-string information available in a block of the LCM object. It is also possible to use subroutine `lcmgcl_c` presented in Section 3.7.3. In the following example, a block is stored in an the heterogeneous list located at address `iplist`. The block is located at the `iset`-th position of the heterogeneous list and has a length equivalent to 5 32-bit words. The information is recovered into the integer array `idata` and transformed into a null-terminated character string `hname` using the `strcut_c` utility:

```
#include "lcm.h"
...
char *namp="...", hname[21];
int_32 iset,idata[5];
lcm *iplist;
iplist=... ;
iset=...;
lcmgdl_c(&iplist,iset,idata);
strcut_c(hname,(char *)idata,20);
```

3.4.7 `lcmpdl_c`

Function used to store a block of data (array of elementary type) into an heterogeneous list. The information is copied from memory towards the LCM object. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed into a LCM object open in **read-only** mode.

```
lcmpdl_c(iplist,iset,ilong,itylcm,data);
```

input parameters:		
iplist	<i>lcm**</i>	address of the heterogeneous list.
iset	<i>int_32</i>	index of the block in the list. The first element of the list is located at index 0.
ilong	<i>int_32</i>	length of the block in 32-bit words. If the array contains N real.64 values, ilong must be set to $2 \times N$.
itylcm	<i>int_32</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
data	<i>int_32*</i>	array of dimension \geq ilong to be copied into the LCM object. Array elements <code>iofset[0]</code> to <code>iofset[ilong-1]</code> must be initialized before the call to <code>lcmpdl_c</code> .

value of the function:	
<i>void</i>	

Function `lcmpdl_c` can be used to store character-string information into an heterogeneous list of a LCM object. In the following example, a character string `hname` is first transformed into an integer array

`idata` using the `strfil_c` utility. This array (block of data) is stored into the LCM object located at address `iplist`, using `lcmpdl_c`. The block is located at the `iset`-th position of the heterogeneous list, has a length equivalent to 5 32-bit words, and a type equal to 3.

```
#include "lcm.h"
...
char *namp="...", hname[20];
int_32 iset, idata[5], it=3, il=5;
lcm *iplist;
iplist=... ;
iset=...;
strfil_c((char *)idata, hname, 20);
lcmpdl_c(&iplist, iset, il, it, idata);
```

3.4.8 `lcmgpl_c`

Function used to recover the memory address of an information block (array of elementary type) from an heterogeneous list, *without* making a copy of the information. Use of this function must respect the following rules:

- If the information is modified after the call to `lcmgpl_c`, a call to `lcmppl_c` must be performed to acknowledge the modification.
- The block `*iofset` should never be released using a deallocation function such as `rlsara_c`, `free`, etc.
- The address `iofset` must never be copied into another variable.

Non respect of these rules may cause execution failure (core dump, segmentation fault, etc) without possibility to throw an exception.

A call to `lcmgpl_c` doesn't cause any modification to the LCM object. The data array information is accessed directly from memory locations `*iofset[0]` to `*iofset[ilong-1]` where `iofset` is the address returned by function `lcmgpl_c`.

```
lcmgpl_c(iplist, iset, iofset);
```

input parameters:		
iplist	<i>lcm**</i>	address of the heterogeneous list.
iset	<i>int_32</i>	index of the block in the list. A call to <code>xabort_c</code> is performed if the block doesn't exist. The first element of the list is located at index 0.

output parameters:		
iofset	<i>int_32**</i>	address of the data array, as returned by <code>setara_c</code> .
value of the function:		
<i>void</i>		

3.4.9 `lcmppl_c`

Function used to store a block of data (array of elementary type) into an heterogeneous list *without* making a copy of the information. If the block already exists, it is replaced; otherwise, it is created. This

operation cannot be performed into a LCM object open in **read-only** mode.

If the **iset-th** component of the heterogeneous list already exists, the address associated with this component is replaced by the new address and the information pointed by the old address is deallocated.

The array containing information stored by **lcmppl.c** must be originally allocated by a call of the form **iofset = setara_c(iling)** or **iofset = (int_32*)malloc(iling*sizeof(int_32))**.

```
lcmppl_c(iplist,iset,iling,itylcm,iofset);
```

input parameters:		
iplist	<i>lcm**</i>	address of the heterogeneous list.
iset	<i>int_32</i>	index of the block in the list. The first element of the list is located at index 0.
iling	<i>int_32</i>	length of the block in 32-bit words.
itylcm	<i>int_32</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
iofset	<i>int_32*</i>	address of the data array, as returned by setara_c . Data elements iofset[0] to iofset[iling-1] must be initialized before the call to lcmppl.c .

value of the function:	
<i>void</i>	

The information block of address **iofset** will automatically be deallocated using function **rlsara_c** at closing time of the LCM object. Situations exist where this block is shared with data structures other than LCM, and where the block must *not* be deallocated by the LCM API. In this case, it is imperative to follow the call to **lcmppl.c** by a call to function **refpush** of the form:

```
refpush(iplist,iofset);
```

3.5 Management of the associative tables and of the heterogeneous lists

These functions permit to create (**lcmsix.c**, **lcmdid.c**, **lcmdil.c**, **lcmlid.c**, **lcmlil.c**) or to access (**lcmsix.c**, **lcmgid.c**, **lcmgil.c**) daughter associative tables or daughter heterogeneous lists. Use of these functions is summarized in the following table:

		daughter	
		associative table	heterogeneous list
father	associative table	lcmdid.c	lcmlid.c
		lcmgid.c	lcmgid.c
	heterogeneous list	lcmdil.c	lcmlil.c
		lcmgil.c	lcmgil.c

3.5.1 *lcmdid.c*

Function used to create or access a daughter associative table included into a father associative table. This operation cannot be performed in a LCM object open in **read-only** mode.

The daughter associative table is created if it doesn't already exist. Otherwise, the existing daughter associative table is accessed. In the latter case, it is recommended to use function `lcmgid_c` which is faster for a simple access and which can be used with LCM object open in **read-only** mode.

```
lcmdid_c(iplist,namp);
```

input parameterss:		
iplist	<i>lcm**</i>	address of the father associative table.
namp	<i>char*</i>	name of the daughter associative table.

value of the function:	
<i>lcm*</i>	address of the daughter associative table.

3.5.2 *lcmlid_c*

Function used to create or access a daughter heterogeneous list included into a father associative table. This operation cannot be performed in a LCM object open in **read-only** mode.

In the following example, a daughter heterogeneous list is created as a block **LIST** into a father associative table. The heterogeneous list contains 5 components. A block of data is stored in each component of the heterogeneous list using `lcmppl_c`:

```
#include "lcm.h"
...
lcm *iplist,*jplist;
int_32 n=5, i ;
...
jplist=lcmlid_c(&iplist,"LIST",n);
for(i=0;i<5;i++) {
    lcmppl_c(&jplist,i,...
}
```

The heterogeneous list capability is implemented through calls to function `lcmlid_c`. Such a call permit the following possibilities:

- the heterogeneous list is created if it doesn't already exist.
- the heterogeneous list is accessed if it already exists *and* if its length is unchanged. In this case, it is recommended to use function `lcmgid_c` which is faster for a simple access and which can be used with LCM object open in **read-only** mode.
- the heterogeneous list is enlarged (components are added) if it already exists *and* if the new length is larger than the preceding one.

```
lcmlid_c(iplist,namp,ilong);
```

input parameterss:		
iplist	<i>lcm**</i>	address of the father associative table.
namp	<i>char*</i>	name of the daughter heterogeneous list.
ilong	<i>int_32</i>	number of components in the daughter heterogeneous list.

value of the function:	
<i>lcm*</i>	address of the daughter heterogeneous list named namp .

3.5.3 *lcmlil_c*

Function used to create or access a daughter heterogeneous list included into a father heterogeneous list. This operation cannot be performed in a LCM object open in **read-only** mode.

In the following example, a daughter heterogeneous list is created as 77-th component of a father heterogeneous list. The heterogeneous list contains 5 components. A block of data is stored in each component of the heterogeneous list using *lcmppl_c*:

```
#include "lcm.h"
...
lcm *iplist,*jplist;
int_32 n=5, i, iset=77 ;
...
jplist=lcmlil_c(&iplist,iset,n);
for(i=0;i<5;i++) {
    lcmppl_c(&jplist,i,...
}
```

The heterogeneous list capability is implemented through calls to function *lcmlil_c*. Such a call permit the following possibilities:

- the heterogeneous list is created if it doesn't already exist.
- the heterogeneous list is accessed if it already exists *and* if its length is unchanged. In this case, it is recommended to use function *lcmgil_c* which is faster for a simple access and which can be used with LCM object open in **read-only** mode.
- the heterogeneous list is enlarged (components are added) if it already exists *and* if the new length is larger than the preceding one.

```
lcmlil_c(iplist,iset,ilong);
```

input parameterss:		
iplist	<i>lcm**</i>	address of the father heterogeneous list.
iset	<i>int_32</i>	index of the daughter heterogeneous list in the father heterogeneous list. The first element of the list is located at index 0.
ilong	<i>int_32</i>	number of components in the daughter heterogeneous list.

value of the function:	
<i>lcm*</i>	address of the daughter heterogeneous list.

3.5.4 *lcmdil_c*

Function used to create or access a daughter associative table included into a father heterogeneous list. This operation cannot be performed in a LCM object open in **read-only** mode.

The daughter associative table is created if it doesn't already exist. Otherwise, the existing daughter associative table is accessed. In the latter case, it is recommended to use function `lcmgil_c` which is faster for a simple access and which can be used with LCM object open in **read-only** mode.

It is a good programming practice to replace a set of N distinct associative tables by a list made of N associative tables, as depicted in Figure 2.

In the example of Figure 2, a set of 5 associative tables, created by `lcmdir_c`:

```
#include "lcm.h"
...
char HDIR[13]
lcm*iplist,*kplist ;
int_32 i;
HDIR[12] = '\0';
for(i=0;i<5;i++) {
    (void)sprintf(HDIR,"GROUP%3d/ 5",i+1);
    kplist=lcmsix_c(&iplist,HDIR);
    lcmppd_c(&kplist,...);
    ...
}
```

are replaced by a list of 5 associative tables, created by `lcmlid_c` and `lcmdil_c`:

```
#include "lcm.h"
...
lcm *iplist,*jplist,*kplist;
int_32 n=5 ;
jplist=lcmlid_c(&iplist,'GROUP',n);
for(i=0;i<5;i++) {
    kplist=lcmdil_c(&jplist,i);
    lcmppd_c(&kplist,...);
}
```

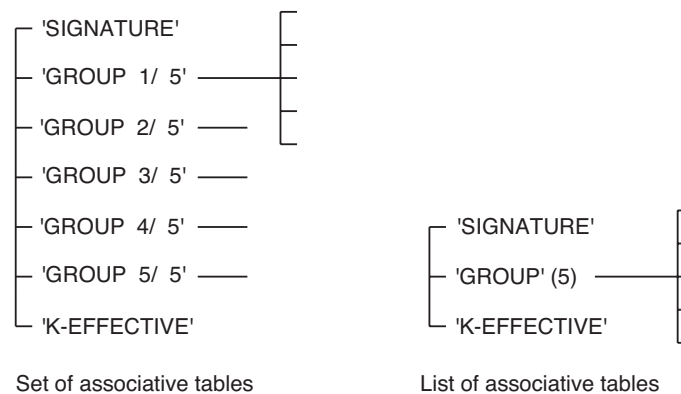


Figure 2: A list of associative tables.

The capability to include associative tables into an heterogeneous list is implemented using the `lcmdil_c` function:

```
lcmdil_c(iplist,iset);
```

input parameterss:		
iplist	<i>lcm**</i>	address of the father heterogeneous list.
iset	<i>int_32</i>	index of the daughter associative table in the father heterogeneous list. The first element of the list is located at index 0.

value of the function:	
<i>lcm*</i>	address of the daughter associative table.

3.5.5 lcmgid_c

Function used to access a daughter associative table or heterogeneous list included into a father associative table.

```
lcmgid_c(iplist,namp);
```

input parameterss:		
iplist	<i>lcm**</i>	address of the father associative table.
namp	<i>char*</i>	name of the daughter associative table or heterogeneous list.

value of the function:	
<i>lcm*</i>	address of the daughter associative table or heterogeneous list. A call to xabort_c is performed if this daughter doesn't extst.

3.5.6 lcmgil_c

Function used to access a daughter associative table or heterogeneous list included into a father heterogeneous list.

```
lcmgil_c(iplist,iset);
```

input parameterss:		
iplist	<i>lcm**</i>	address of the father heterogeneous list.
iset	<i>int_32</i>	index of the daughter associative table or heterogeneous list in the father heterogeneous list. The first element of the list is located at index 0.

value of the function:	
<i>lcm*</i>	address of the daughter associative table or heterogeneous list. A call to xabort_c is performed if this daughter doesn't extst.

3.5.7 lcmsix_c

Function used to move across the hierarchical structure of a LCM object made of associative tables. Using this function, there is no need to remember the names of the father (grand-father, etc.) associative tables. If a daughter associative table doesn't exist and if the LCM object is open on creation or modification mode, the daughter associative table is created. A daughter associative table cannot be created if the LCM object is open in **read-only** mode.

Function `lcmsix_c` is deprecated, as `lcmdid_c` offers a more elegant way to perform the same operation. However, `lcmsix_c` is kept available in the LCM API for historical reasons.

```
lcmsix_c(iplist,namp,iact);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table before the call to <code>lcmsix_c</code> .
namp	<i>char**</i>	name of the daughter associative table if <code>iact=1</code> . This parameter is not used if <code>iact=0</code> or <code>iact=2</code> .
iact	<i>int_32</i>	type of move: =0 return towards the root directory of the LCM object; =1 move towards the daughter associative table (create it if it doesn't exist); =2 return towards the father associative table.

output parameters:		
iplist	<i>lcm**</i>	address of the associative table after the call to <code>lcmsix_c</code> .
value of the function:		
<i>void</i>		

3.6 LCM utility functions

3.6.1 lcmlib_c

Function used to print (towards `stdout`) the content of the active directory of an associative table or heterogeneous list.

```
lcmlib_c(iplist);
```

input parameter:		
iplist	<i>lcm**</i>	address of the associative table or of the heterogeneous list.

value of the function:		
<i>void</i>		

3.6.2 lcmequ_c

Function used to perform a deep-copy of the information contained in an associative table (address `iplis1`) towards another associative table (address `iplis2`). Note that the second associative table (address `iplis2`) is modified but not created by `lcmequ_c`.

```
lcmequ_c(iplis1,iplis2);
```

input parameter:		
iplis1	<i>lcm**</i>	address of the existing associative table or of the heterogeneous list (accessed in read-only mode).

output parameters:		
iplis2	<i>lcm**</i>	address of the associative table or of the heterogeneous list, modified by lcmequ_c .
value of the function:		
<i>void</i>		

3.6.3 *lcmexp_c*

Function used to export (or import) the content of an associative table towards (or from) a sequential file. The sequential file can be in binary or **ascii** format.

The export of information starts from the active directoty. Note that *lcmexp_c* is basically a serialization algorithm based on the contour algorithm.

```
lcmexp_c(iplist,impx,file,imode,idir);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table or of the heterogeneous list to be exported (or imported).
impx	<i>int_32</i>	print parameter (equal to 0 for no print).
file	<i>FILE*</i>	sequential file.
imode	<i>int_32</i>	=1 binary sequential file; =2 ASCII sequential file.
idir	<i>int_32</i>	=1 to export; =2 to import.

value of the function:	
<i>void</i>	

3.7 Using variable-length string arrays

The following functions are implemented using the C functions of the preceding sections. They permit the use of variable-length string arrays, a capability not yet available with the Fortran LCM API.

		type of operation	
		put	get
father	associative table	lcmpcd_c	lcmgcd_c
	heterogeneous list	lcmpcl_c	lcmgcl_c

3.7.1 *lcmgcd_c*

Function used to recover a variable-length string array from a block of data stored in an associative table.


```
lcmgcd_c(iplist,namp,hdata);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the variable-length string array to recover. A call to <code>xabort_c</code> is performed if the block does't exist.

output parameters:		
hdata	<i>char**</i>	variable-length string array of dimension \geq <code>ilong</code> . The memory space required to represent the string array is allocated by <code>lcmgcd_c</code> .
value of the function:		
<i>void</i>		

3.7.2 `lcmpcd_c`

Function used to store a variable-length string array into a block of data stored in an associative table. If the block of data already exists, it is updated; otherwise, it is created. This operation cannot be performed in a LCM object open in `read-only` mode.

```
lcmpcd_c(iplist,namp,ilong,hdata);
```

input parameters:		
iplist	<i>lcm**</i>	address of the associative table.
namp	<i>char*</i>	name of the variable-length string array to store.
ilong	<i>int_32</i>	number of components in the variable-length string array.
hdata	<i>char**</i>	array of dimension \geq <code>ilong</code> to be copied in the LCM object.

value of the function:		
<i>void</i>		

Example:

```
#include "lcm.h"
...
lcm *iplist;
int_32 i, ilong = 5;
char *hdata1[ilong],*hdata2[ilong];

hdata1[0] = "string1";
hdata1[1] = "  string2";
hdata1[2] = "    string3";
hdata1[3] = "      string4";
hdata1[4] = "        string5";
for (i=0;i<ilong;i++) {
    printf("i=%d string='%s' size=%d\n",i,hdata1[i],strlen(hdata1[i]));
}

lcmop_c(&iplist,"mon_dict",0,1,2);
```

```

/* Store the information */
lcmgcd_c(&iplist,"node1",ilong,hdata1);

/* Recover the information */
lcmgcd_c(&iplist,"node1",hdata2);

for (i=0;i<ilong;i++) {
    printf("in table i=%d string='%s' size=%d\n",i,hdata2[i],strlen(hdata2[i]));
}
for (i=0;i<ilong;i++) free(hdata2[i]);
lcmcl_c(&iplist,2);

```

3.7.3 *lcmgcl_c*

Function used to recover a variable-length string array from a block of data stored in an heterogeneous list.

```
lcmgcl_c(iplist,namp,hdata);
```

input parameters:		
iplist	<i>lcm**</i>	address of the heterogeneous list.
iset	<i>int_32</i>	index of the variable-length string array in the heterogeneous list. A call to <code>xabort_c</code> is performed if the component doesn't exist. The first element of the list is located at index 0.

output parameters:		
hdata	<i>char**</i>	variable-length string array of dimension \geq <code>ilong</code> . The memory space required to represent the string array is allocated by <code>lcmgcl_c</code> .
value of the function:		
<i>void</i>		

3.7.4 *lcmpcl_c*

Function used to store a variable-length string array into a block of data stored in an heterogeneous list. If the block of data already exists, it is updated; otherwise, it is created. This operation cannot be performed in a LCM object open in **read-only** mode.

```
lcmpcl_c(iplist,iset,ilong,hdata);
```

input parameters:		
iplist	<i>lcm**</i>	address of the heterogeneous list.
iset	<i>int_32</i>	index of the variable-length string array in the heterogeneous list. The first element of the list is located at index 0.
ilong	<i>int_32</i>	number of components in the variable-length string array .
hdata	<i>char**</i>	array of dimension \geq <code>ilong</code> to be copied in the LCM object.

value of the function:	
<i>void</i>	

Example:

```
#include "lcm.h"
...
lcm *iplist, *jplist;
int_32 i, ilong = 5;
char *hdata1[ilong],*hdata2[ilong];

hdata1[0] = "string1";
hdata1[1] = "  string2";
hdata1[2] = "    string3";
hdata1[3] = "      string4";
hdata1[4] = "        string5";
for (i=0;i<ilong;i++) {
    printf("i=%d string='%s' size=%d\n",i,hdata1[i],strlen(hdata1[i]));
}

lcmop_c(&iplist,"mon_dict",0,1,2);

/* Creation of the heterogeneous list */
jplist = lcmlid_c(&iplist,"node2",77);

/* Store the information */
lcmpcl_c(&jplist,4,ilong,hdata1);

/* Recover the information */
lcmgcl_c(&jplist,4,hdata2);

for (i=0;i<ilong;i++) {
    printf("in list i=%d string='%s' size=%d\n",i,hdata2[i],strlen(hdata2[i]));
}
for (i=0;i<ilong;i++) free(hdata2[i]);

lcmcl_c(&iplist,2);
```

3.8 Dynamic allocation of the elementary blocks of data

3.8.1 *setara_c*

Function used to allocate a block of data for storing a memory-resident `int_32` data array. Function `setara_c` is a simple wrapper for `malloc` standard library function. If the operating system fails to allocate the memory, a call to `xabort_c` is performed.

```
setara_c(ilong);
```

input parameter:		
<code>ilong</code>	<i>int_32</i>	length of the block of data to allocate in unit of 32-bit words.

value of the function:	
<i>int_32*</i>	address of the allocated block of data.

3.8.2 *rlsara_c*

Function used to deallocate a memory-resident block of data previously allocated by **setara_c**. The implementation of **rlsara_c** in ANSI C is based on the **free** standard library function. If the operating system fails to deallocate the memory, a call to **xabort_c** is performed.

```
rlsara_c(iofset);
```

input parameter:		
iofset	<i>int_32*</i>	address of the block of data to deallocate. This value <i>must</i> have been allocated by a previous call to setara_c .

value of the function:	
<i>void</i>	

3.9 Abnormal termination of the execution

3.9.1 *xabort_c*

Function used to cause the program termination. A message describing the conditions of the termination is printed.

It is important to use this function to abort a program instead of using the **exit()** function of the standard library. The **xabort_c** function can be used to implement *exception treatment* in situations where the application software is driven by a multi-physics system.

If an abnormal termination occurs, the **xabort_c** function is called as

```
xabort_c("sub001: execution failure.");
```

```
xabort_c(hsmg);
```

input parameter:		
hsmg	<i>char*</i>	message describing the conditions of the abnormal termination.

value of the function:	
<i>void</i>	

4 The ISO Fortran LCM API

The ISO Fortran LCM API is a set of Fortran-2003 wrapper subroutines or functions programmed around the ANSI-C functions of the LCM API. This implementation is using the C interoperability capabilities normalized by ISO and available in the Fortran-2003 compilers. All the subroutines and functions presented in this section are ISO-standard and 64-bit clean.

Each LCM object has a *root associative table* from which the complete object is constructed.

Any subroutines or functions using the Fortran LCM API must include a `USE` statement of the form

```
USE GANLIB
```

The address of a LCM object is a `TYPE(C_PTR)` variable declared as

```
TYPE(C_PTR) :: IPLIST
```

This intrinsic type is defined by the `USE GANLIB` statement. Very few operations are permitted on `C_PTR` variables. A `C_PTR` variable can be nullified by writing

```
IPLIST=C_NULL_PTR
```

and a `C_PTR` variable can be checked for association with actual data using

```
IF(C_ASSOCIATED(IPLIST)) THEN
```

4.1 Opening, closing and validation of LCM objects

4.1.1 LCMOP

Open an LCM object (either memory resident or persistent). Obtain the address of the LCM object if it is created. Note that CLE-2000 is responsible to perform the calls to `LCMOP` for the LCM objects that are used as parameters of a CLE-2000 module. The use of `LCMOP` is generally restricted to the use of temporary LCM objects created within a CLE-2000 module.

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

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the LCM object if <code>IMP=1</code> or <code>imp=2</code> . <code>IPLIST</code> corresponds to the address of the root associative table.
NAMP	<i>CHARACTER*12</i>	name of the LCM object if <code>IMP=0</code> .
IMP	<i>INTEGER</i>	=0 to create a new LCM object ; =1 to modify an existing LCM object; =2 to access an existing LCM object in read-only mode.
MEDIUM	<i>INTEGER</i>	=1 to use a memory-resident LCM object; =2 to use an XSM file to store the LCM object.
IMPX	<i>INTEGER</i>	print parameter. Equal to zero to suppress all printings.

output parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of an LCM object if <code>IMP=0</code> .
NAMP	<i>CHARACTER*12</i>	name of the LCM object if <code>IMP=1</code> or <code>IMP=2</code> .

4.1.2 LCMCL

Close an LCM object (either memory resident or persistent). Note that CLE-2000 is responsible to perform the calls to LCMCL for the LCM objects that are used as parameters of a CLE-2000 module. The use of LCMCL is generally restricted to the use of temporary LCM objects created within a CLE-2000 module.

A LCM object can only be closed if IPLIST points towards its root directory.

CALL LCMCL(IPLIST,IACT)

input parameters:		
IPLIST	TYPE(C_PTR)	address of the LCM object (address of the root directory of the LCM object).
IACT	INTEGER	=1 close the LCM object without destroying it; =2 and destroying it

output parameter:		
IPLIST	TYPE(C_PTR)	IPLIST=0 indicates that the LCM object is closed and destroyed. A memory-resident LCM object keeps the same address during its complete existence. A persistent LCM object is associated to an XSM file and is represented by a different value of IPLIST each time it is reopened.

4.1.3 LCMVAL

Subroutine to validate a single block of data in a LCM object or the totality of the LCM object, starting from the address of an associative table. This function has no effect if the object is persistent. The validation consists to verify the connections between the elements of the LCM object, to verify that each element of the object is defined and to check for possible memory corruptions. If an error is detected, the following message is issued:

LCMVAL: BLOCK xxx OF THE TABLE yyy HAS BEEN OVERWRITTEN.

This function is called as

CALL LCMVAL(IPLIST,NAMP)

input parameters:		
IPLIST	TYPE(C_PTR)	address of the associative table or of the heterogeneous list.
NAMP	CHARACTER*12	name of the block to validate in the associative table. If NAMP=' ', all the blocks in the associative table are verified in a recursive way.

4.2 Interrogation of LCM objects

The data structures in an LCM object are self-described. It is therefore possible to interrogate them in order to know their characteristics.

		type of interrogation	
		father structure	information block
father	associative table	LCMINF LCMNXT	LCMLEN
	heterogeneous list	LCMINF	LCMLEL

4.2.1 LCMLN

Subroutine used to recover the length and type of an information block stored in an associative table (either memory-resident or persistent). The length is the number of elements in a daughter heterogeneous list or the number of consecutive 32-bit memory words required to store an array of elementary type. As an example, the length required to store an array of 64-bit real numbers is twice its dimension.

CALL LCMLN(IPLIST,NAMP,ILONG,ITYLCM)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the block.

output parameters:		
ILONG	<i>INTEGER</i>	length of the block in 32-bit words. =-1 for a daughter associative table; = <i>N</i> for a daughter heterogeneous list containing <i>N</i> components; =0 if the block doesn't exist.
ITYLCM	<i>INTEGER</i>	type of information. =0 associative table; =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =10 heterogeneous list; =99 undefined (99 is returned if the block doesn't exist).

4.2.2 LCMINF

Subroutine used to recover general information about a LCM object.

CALL LCMINF(IPLIST,NAMLCM,NAMMY,EMPTY,ILONG,LCM)

input parameter:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table or of the heterogeneous list.

output parameters:		
NAMLCM	<i>CHARACTER*12</i>	name of the LCM object.
NAMMY	<i>CHARACTER*12</i>	name of the associative table at address IPLIST. = '/' if the associative table is the root of the LCM object; = ' ' if the associative table is an heterogeneous list component.
EMPTY	<i>LOGICAL</i>	logical variable set to .true. if the associative table is empty or set to .false. otherwise.
ILONG	<i>INTEGER</i>	= -1: IPLIST is an associative table; > 0: number of components in the heterogeneous list IPLIST
LCM	<i>LOGICAL</i>	logical variable set to .true. if information is memory-resident or set to .false. if information is persistent (stored in an XSM file).

4.2.3 LCMNXT

Subroutine used to find the name of the next block of data in an associative table. Use of LCMNXT is forbidden if the associative table is empty. The order of names is arbitrary. The search cycle indefinitely.

CALL LCMNXT(IPLIST,NAMP)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of an existing block. NAMP=' ' can be used to obtain a first name to initiate the search.

output parameter:		
NAMP	<i>CHARACTER*12</i>	name of the next block. A call to XABORT is performed if the associative table is empty.

4.2.4 LCMLEL

Subroutine used to recover the length and type of an information block stored in an heterogeneous list (either memory-resident or persistent). The length is the number of elements in a daughter heterogeneous list or the number of consecutive 32-bit memory words required to store an array of elementary type. As an example, the length required to store an array of 64-bit real numbers is twice its dimension.

CALL LCMLEL(IPLIST,ISET,ILONG,ITYLCM)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the heterogeneous list.
ISET	<i>INTEGER</i>	index of the block in the list. The first element of the list is located at index 1.

output parameters:		
ILONG	INTEGER	length of the block in 32-bit words. =0 if the block does't exist.
ITYLCM	INTEGER	type of information. =0 associative table; =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =10 heterogeneous list; =99 undefined (99 is returned if the block does't exist).

4.3 Management of the array of elementary type

Management of the array of elementary type can be performed *with* copy of the data (LCMPUT, LCMGET, LCMPTL or LCMGDL) or *without* copy (LCMPPD, LCMGPD, LCMPL or LCMGPL).

		type of operation	
		put	get
father	associative table	LCMPUT LCMPPD	LCMGET LCMGPD
	heterogeneous list	LCMPTL LCMPL	LCMGDL LCMGPL

4.3.1 LCMGET

Subroutine used to recover an information block (array of elementary type) from an associative table and to copy this data into memory.

CALL LCMGET(IPLIST,NAMP,DATA)

input parameters:		
IPLIST	TYPE(C_PTR)	address of the associative table.
NAMP	CHARACTER*12	name of the block to recover. A call to XABORT is performed if the block does't exist.

output parameter:		
DATA	CLASS(*)	array of dimension \geq ILONG in which the block is copied.

Subroutine LCMGET can be used to recover character-string information available in a block of the LCM object. It is also possible to use subroutine LCMGCD presented in Section 4.7.1. In the following example, a block is stored in an associative table located at address IPLIST. The block has a name NAMP and a length equivalent to 5 32-bit words. The information is recovered into the integer array IDATA and transformed into a **character*20** variable named HNAME using an internal WRITE statement:

```

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

```

4.3.2 LCMPUT

Subroutine used to store a block of data (array of elementary type) into an associative table. The information is copied from memory towards the LCM object. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed into a LCM object open in **read-only** mode.

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

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the block.
ILONG	<i>INTEGER</i>	length of the block in 32-bit words. If the array contains N real.64 values, <code>ilong</code> must be set to $2 \times N$.
ITYLCM	<i>INTEGER</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
DATA	<i>CLASS(*)</i>	array of dimension \geq ILONG to be copied into the LCM object. Array elements DATA(1) to DATA(ILONG) must be initialized before the call to LCMPUT.

Subroutine LCMPUT can be used to store character-string information in an associative table of a LCM object. It is also possible to use function LCMPCD presented in Section 4.7.2. In the following example, a character string HNAME is first transformed into an integer array IDATA using an internal READ statement. This array (block of data) is stored into the LCM object located at address IPLIST, using LCMPUT. The block has a name NAMP, a length equivalent to 5 32-bit words, and a type equal to 3.

```

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

```

4.3.3 LCMGPD

Subroutine used to recover the TYPE(C_PTR) address of an information block (array of elementary type) from an associative table, *without* making a copy of the information. Use of this subroutine must respect the following rules:

- If the information is modified after the call to LCMGPD, a call to LCMPPD must be performed to acknowledge the modification.
- The block pointed by IOFSET should never be released using a deallocation function such as RLSARA, **deallocate**, etc.
- The variable IOFSET must never be copied into another variable.

Non respect of these rules may cause execution failure (core dump, segmentation fault, etc) without possibility to throw an exception.

Subroutine LCMGPD implements direct *pinning* on LCM data structures. It represents an advanced capability of the LCM API and should only be used in situations where the economy of computer resources is a critical issue. The C_PTR address is the ANSI C pointer of a block of information made available into a Fortran program. If IOFSET is a C_PTR address, the useful information is accessed in a Fortran variable IDATA set using

```
USE GANLIB
...
TYPE(C_PTR) :: IOFSET
INTEGER, POINTER, DIMENSION(:) :: IDATA
...
CALL LCMGPD(IPLIST,NAMP,IOFSET)
CALL C_F_POINTER(IOFSET,IDATA, (/ ILONG /))
```

The useful information is therefore accessed in memory locations IDATA(1) to IDATA(ILONG).

A call to LCMGPD doesn't cause any modification to the LCM object.

```
CALL LCMGPD(IPLIST,NAMP,IOFSET)
```

input parameters:		
IPLIST	TYPE(C_PTR)	address of the associative table.
NAMP	CHARACTER*12	name of the block to recover. A call to XABORT is performed if the block doesn't exist.

output parameter:		
IOFSET	TYPE(C_PTR)	C_PTR address of the information.

4.3.4 LCMPPD

Subroutine used to store a block of data (array of elementary type) into an associative table *without* making a copy of the information. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed into a LCM object open in **read-only** mode.

If a block named NAMP already exists in the associative table, the address associated with NAMP is replaced by the new address and the information pointed by the old address is deallocated.

Subroutine LCMPPD implements direct *pinning* on LCM data structures. It represents an advanced capability of the LCM API and should only be used in situations where the economy of computer resources is a critical issue. The memory block stored by LCMPPD must be previously allocated by a call to LCMARA of the form

```
IOFSET=LCMARA(ILONG)
```

so that the useful information is accessed in a Fortran variable IDATA set using a C_F_POINTER function:

```
USE GANLIB
...
TYPE(C_PTR) :: IOFSET
```

```

INTEGER, POINTER, DIMENSION(:) :: IDATA
...
IOFSET = LCMARA(ILONG)
CALL C_F_POINTER(IOFSET, IDATA, (/ ILONG /))
...
CALL LCMPPD(IPLIST, NAMP, ILONG, ITYLCM, IOFSET)

```

The useful information is therefore accessed in memory locations `IDATA(1)` to `IDATA(ILONG)`. There is no need to declare `LCMARA` as an external function; this declaration is included in the module set by the `USE GANLIB` statement.

```
CALL LCMPPD(IPLIST, NAMP, ILONG, ITYLCM, IOFSET)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the block.
ILONG	<i>INTEGER</i>	length of the block in 32-bit words.
ITYLCM	<i>INTEGER</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
IOFSET	<i>TYPE(C_PTR)</i>	C_PTR address of the information. Data elements pointed by IOFSET must be initialized before the call to LCMPPD.

output parameter:		
IOFSET	<i>TYPE(C_PTR)</i>	IOFSET=C_NULL_PTR to indicate that the information previously pointed by IOFSET is now managed by LCM.

4.3.5 LCMDEL

Subroutine used to erase an information block or a daughter heterogeneous list stored in a memory-resident associative table. Subroutine `LCMDEL` *cannot* be used with persistent LCM objects.

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

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the block to erase.

4.3.6 LCMGDL

Subroutine used to recover an information block (array of elementary type) from an heterogeneous list and to copy this data into memory.

```
CALL LCMGDL(IPLIST, ISET, DATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the heterogeneous list.
ISSET	<i>INTEGER</i>	index of the block in the heterogeneous list. A call to XABORT is performed if the block does't exist. The first element of the list is located at index 1.

output parameter:		
DATA	<i>CLASS(*)</i>	array of dimension \geq ILONG in which the block is copied.

Subroutine **LCMGDL** can be used to recover character-string information available in a block of the LCM object. It is also possible to use subroutine **LCMGCL** presented in Section 4.7.3. In the following example, a block is stored in an heterogeneous list located at address **IPLIST**. The block is located at the **ISSET**-th position of the heterogeneous list and has a length equivalent to 5 32-bit words. The information is recovered into the integer array **IDATA** and transformed into a **character*20** variable named **HNAME** using an internal **WRITE** statement:

```

USE GANLIB
...
CHARACTER HNAME*20
INTEGER IDATA(5)
TYPE(C_PTR) IPLIST
...
IPLIST=...
ISSET=...
CALL LCMGDL(IPLIST,ISSET,IDATA)
WRITE(HNAME,'(5A4)') (IDATA(I),I=1,5)

```

4.3.7 LCMPDL

Subroutine used to store a block of data (array of elementary type) into an heterogeneous list. The information is copied from memory towards the LCM object. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed into a LCM object open in **read-only** mode.

```
CALL LCMPDL(IPLIST,ISSET,ILONG,ITYLCM,DATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the heterogeneous list.
ISSET	<i>INTEGER</i>	index of the block in the list. The first element of the list is located at index 1.
ILONG	<i>INTEGER</i>	length of the block in 32-bit words. If the array contains N real.64 values, ILONG must be set to $2 \times N$
ITYLCM	<i>INTEGER</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
DATA	<i>CLASS(*)</i>	array of dimension \geq ILONG to be copied into the LCM object. Array elements IOFSET(1) to IOFSET(ILONG) must be initialized before the call to LCMPDL .

Subroutine LCMPDL can be used to store character-string information into an heterogeneous list of a LCM object. In the following example, a character string HNAME is first transformed into an integer array IDATA using an internal READ statement. This array (block of data) is stored into the LCM object located at address IPLIST, using LCMPDL. The block is located at the ISET-th position of the heterogeneous list, has a length equivalent to 5 32-bit words, and a type equal to 3.

```

USE GANLIB
...
CHARACTER HNAME*20
INTEGER IDATA(5)
TYPE(C_PTR) IPLIST
...
IPLIST=...
ISET=...
READ(HNAME,'(5A4)')(IDATA(I),I=1,5)
CALL LCMPDL(IPLIST,ISET,5,3,IDATA)

```

4.3.8 LCMGPL

Subroutine used to recover the TYPE(C_PTR) address of an information block (array of elementary type) from an heterogeneous list, *without* making a copy of the information. Use of this subroutine must respect the following rules:

- If the information is modified after the call to LCMGPL, a call to LCMPPPL must be performed to acknowledge the modification.
- The block pointed by IOFSET should never be released using a deallocation function such as RLSARA, deallocate, etc.
- The variable IOFSET must never be copied into another variable.

Non respect of these rules may cause execution failure (core dump, segmentation fault, etc) without possibility to throw an exception.

Subroutine LCMGPL implements direct *pinning* on LCM data structures. It represents an advanced capability of the LCM API and should only be used in situations where the economy of computer resources is a critical issue. The C_PTR address is the ANSI C pointer of a block of information made available into a Fortran program. If IOFSET is a C_PTR address, the useful information is accessed in a Fortran variable IDATA set using

```

USE GANLIB
...
TYPE(C_PTR) :: IOFSET
INTEGER, POINTER, DIMENSION(:) :: IDATA
...
CALL LCMGPL(IPLIST,ISET,IOFSET)
CALL C_F_POINTER(IOFSET,IDATA, (/ ILONG /))

```

The useful information is therefore accessed in memory locations IDATA(1) to IDATA(ILONG).

A call to LCMGPL doesn't cause any modification to the LCM object.

```
CALL LCMGPL(IPLIST,ISET,IOFSET)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the heterogeneous list.
ISET	<i>INTEGER</i>	index of the block in the list. A call to XABORT is performed if the block doesn't exist. The first element of the list is located at index 1.

output parameter:		
IOFSET	<i>TYPE(C_PTR)</i>	C_PTR address of the information.

4.3.9 LCMPPPL

Subroutine used to store a block of data (array of elementary type) into an heterogeneous list *without* making a copy of the information. If the block already exists, it is replaced; otherwise, it is created. This operation cannot be performed into a LCM object open in **read-only** mode.

If the ISET-th component of the heterogeneous list already exists, the address associated with this component is replaced by the new address and the information pointed by the old address is deallocated.

Subroutine LCMPPPL implements direct *pinning* on LCM data structures. It represents an advanced capability of the LCM API and should only be used in situations where the economy of computer resources is a critical issue. The memory block stored by LCMPPPL must be previously allocated by a call to LCMARA of the form

```
IOFSET=LCMARA(ILONG)
```

so that the useful information is accessed in a Fortran variable IDATA set using a C_F_POINTER function:

```
USE GANLIB
...
TYPE(C_PTR) :: IOFSET
INTEGER, POINTER, DIMENSION(:) :: IDATA
...
IOFSET = LCMARA(ILONG)
CALL C_F_POINTER(IOFSET, IDATA, (/ ILONG /))
...
CALL LCMPPPL(IPLIST, ISET, ILONG, ITYLCM, IOFSET)
```

The useful information is therefore accessed in memory locations IDATA(1) to IDATA(ILONG). There is no need to declare LCMARA as an external function; this declaration is included in the module set by the **USE GANLIB** statement.

```
CALL LCMPPPL(IPLIST, ISET, ILONG, ITYLCM, IOFSET)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the heterogeneous list.
ISET	<i>INTEGER</i>	index of the block in the list. The first element of the list is located at index 1.
ILONG	<i>INTEGER</i>	length of the block in 32-bit words.
ITYLCM	<i>INTEGER</i>	type of information. =1 32-bit integer; =2 32-bit real; =3 character*4 data; =4 64-bit real; =5 32-bit logical; =6 64-bit complex; =99 undefined.
IOFSET	<i>TYPE(C_PTR)</i>	C_PTR address of the information. Data elements pointed by IOFSET must be initialized before the call to LCMPPPL.

output parameter:		
IOFSET	<i>INTEGER</i>	IOFSET=C_NULL_PTR to indicate that the information previously pointed by IOFSET is now managed by LCM.

4.4 Management of the associative tables and of the heterogeneous lists

These functions permit to create (LCMSIX, LCMDID, LCMDIL, LCMLID, LCMLIL) or to access (LCMSIX, LCMGID, LCMGIL) daughter associative tables or daughter heterogeneous lists. There is no need to declare these functions as external functions; this declaration is included in the module set by the `USE GANLIB` statement. Use of these functions is summarized in the following table:

		daughter	
		associative table	heterogeneous list
father	associative table	LCMDID	LCMLID
		LCMGID	LCMGID
	heterogeneous list	LCMDIL	LCMLIL
		LCMGIL	LCMGIL

4.4.1 LCMDID

Function used to create or access a daughter associative table included into a father associative table. This operation cannot be performed in a LCM object open in **read-only** mode.

The daughter associative table is created if it doesn't already exist. Otherwise, the existing daughter associative table is accessed. In the latter case, it is recommended to use function LCMGID which is faster for a simple access and which can be used with LCM object open in **read-only** mode.

JPLIST=LCMDID(IPLIST,NAMP)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the father associative table.
NAMP	<i>CHARACTER*12</i>	name of the daughter associative table.

output parameter:		
JPLIST	<i>TYPE(C_PTR)</i>	address of the daughter associative table.

4.4.2 LCMLID

Function used to create or access a daughter heterogeneous list included into a father associative table. This operation cannot be performed in a LCM object open in **read-only** mode.

In the following example, a daughter heterogeneous list is created as a block `LIST` into a father associative table. The heterogeneous list contains 5 components. A block of data is stored in each component of the heterogeneous list using `LCMPDL`:

```
USE GANLIB
...
TYPE(C_PTR) :: IPLIST, JPLIST
```



```

...
JPLIST=LCMLID(IPLIST,'LIST',5)
DO I=1,5
    CALL LCMPDL(JPLIST,I,...
...
ENDDO

```

The heterogeneous list capability is implemented through calls to function LCMLID. Such a call permit the following possibilities:

- the heterogeneous list is created if it doesn't already exist.
- the heterogeneous list is accessed if it already exists *and* if its length is unchanged. In this case, it is recommended to use function LCMGID which is faster for a simple access and which can be used with LCM object open in **read-only** mode.
- the heterogeneous list is enlarged (components are added) if it already exists *and* if the new length is larger than the preceding one.

JPLIST=LCMLID(IPLIST,NAMP,ILONG)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the father associative table.
NAMP	<i>CHARACTER*12</i>	name of the daughter heterogeneous list.
ILONG	<i>INTEGER</i>	number of components in the daughter heterogeneous list.

output parameter:		
JPLIST	<i>INTEGER</i>	address of the daughter heterogeneous list named NAMP.

4.4.3 LCMLIL

Function used to create or access a daughter heterogeneous list included into a father heterogeneous list. This operation cannot be performed in a LCM object open in **read-only** mode.

In the following example, a daughter heterogeneous list is created as 77-th component of a father heterogeneous list. The heterogeneous list contains 5 components. A block of data is stored in each component of the heterogeneous list using LCMPDL:

```

USE GANLIB
...
TYPE(C_PTR) :: IPLIST, JPLIST
...
JPLIST=LCMLIL(IPLIST,77,5)
DO I=1,5
    CALL LCMPDL(JPLIST,I,...
...
ENDDO

```

The heterogeneous list capability is implemented through calls to function LCMLIL. Such a call permit the following possibilities:

- the heterogeneous list is created if it doesn't already exist.
- the heterogeneous list is accessed if it already exists *and* if its length is unchanged. In this case, it is recommended to use function `LCMGIL` which is faster for a simple access and which can be used with LCM object open in **read-only** mode.
- the heterogeneous list is enlarged (components are added) if it already exists *and* if the new length is larger than the preceding one.

JPLIST=LCMLIL(IPLIST,ISET,ILONG)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the father heterogeneous list.
ISET	<i>INTEGER</i>	index of the daughter heterogeneous list in the father heterogeneous list. The first element of the list is located at index 1.
ILONG	<i>INTEGER</i>	number of components in the daughter heterogeneous list.

output parameter:		
JPLIST	<i>TYPE(C_PTR)</i>	address of the daughter heterogeneous list.

4.4.4 LCMDIL

Function used to create or access a daughter associative table included into a father heterogeneous list. This operation cannot be performed in a LCM object open in **read-only** mode.

The daughter associative table is created if it doesn't already exist. Otherwise, the existing daughter associative table is accessed. In the latter case, it is recommended to use function `LCMGIL` which is faster for a simple access and which can be used with LCM object open in **read-only** mode.

It is a good programming practice to replace a set of N distinct associative tables by a list made of N associative tables, as depicted in Figure 2.

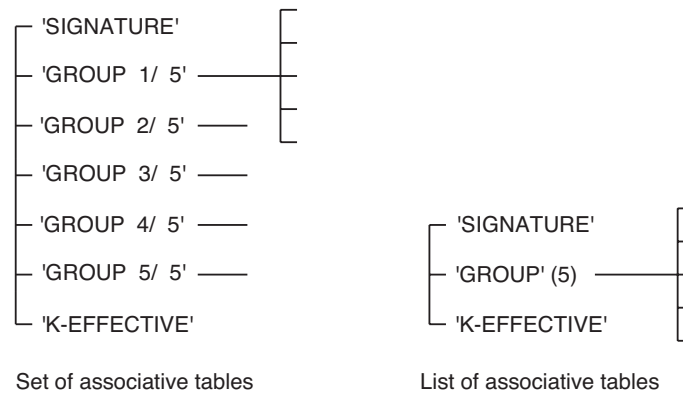


Figure 3: A list of associative tables.

In the example of Figure 3, a set of 5 associative tables, created by LCMDID:

```

USE GANLIB
...
TYPE(C_PTR) :: IPLIST, JPLIST
CHARACTER HDIR*12
...
DO I=1,5
WRITE(HDIR,'(5HGROUP,I3,4H/ 5)') I
JPLIST=LCMDID(IPLIST,HDIR)
CALL LCMPUT(JPLIST,...
...
ENDDO

```

are replaced by a list of 5 associative tables, created by LCMLID and LCMDIL:

```

USE GANLIB
...
TYPE(C_PTR) :: IPLIST, JPLIST, KPLIST
...
JPLIST=LCMLID(IPLIST,'GROUP',5)
DO I=1,5
KPLIST=LCMDIL(JPLIST,I)
CALL LCMPUT(KPLIST,...
...
ENDDO

```

The capability to include associative tables into an heterogeneous list is implemented using the LCMDIL function:

JPLIST=LCMDIL(IPLIST,ISET)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the father heterogeneous list.
ISET	<i>INTEGER</i>	index of the daughter associative table in the father heterogeneous list. The first element of the list is located at index 1.

output parameter:		
JPLIST	<i>TYPE(C_PTR)</i>	address of the daughter associative table.

4.4.5 LCMGID

Function used to access a daughter associative table or heterogeneous list included into a father associative table.

JPLIST=LCMGID(IPLIST,NAMP)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the father associative table.
NAMP	<i>CHARACTER*12</i>	name of the daughter associative table or heterogeneous list.

output parameter:		
JPLIST	<i>TYPE(C_PTR)</i>	address of the daughter associative table or heterogeneous list. A call to XABORT is performed if this daughter doesn't extst.

4.4.6 LCMGIL

Function used to access a daughter associative table or heterogeneous list included into a father heterogeneous list.

JPLIST=LCMGIL(IPLIST,ISET)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the father heterogeneous list.
ISET	<i>INTEGER</i>	index of the daughter associative table or heterogeneous list in the father heterogeneous list. The first element of the list is located at index 1.

output parameter:		
JPLIST	<i>TYPE(C_PTR)</i>	address of the daughter associative table or heterogeneous list. A call to XABORT is performed if this daughter doesn't extst.

4.4.7 LCMSIX

Function used to move across the hierarchical structure of a LCM object made of associative tables. Using this function, there is no need to remember the names of the father (grand-father, etc.) associative tables. If a daughter associative table doesn't exist and if the LCM object is open on creation or modification mode, the daughter associative table is created. A daughter associative table cannot be created if the LCM object is open in **read-only** mode.

Function LCMSIX is deprecated, as LCMDID offers a more elegant way to perform the same operation. However, LCMSIX is kept available in the LCM API for historical reasons.

CALL LCMSIX(IPLIST,NAMP,IACT)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table before the call to LCMSIX.
NAMP	<i>CHARACTER*12</i>	name of the daughter associative table if iact=1 . This parameter is not used if iact=0 or iact=2 .
IACT	<i>INTEGER</i>	type of move: =0 return towards the root directory of the LCM object; =1 move towards the daughter associative table (create it if it doesn't exist); =2 return towards the father associative table.

output parameter:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table after the call to LCMSIX.

4.5 LCM utility functions

4.5.1 LCMLIB

Function used to print (towards `stdout`) the content of the active directory of an associative table or heterogeneous list.

CALL LCMLIB(IPLIST)

input parameter:		
IPLIST	TYPE(C_PTR)	address of the associative table or of the heterogeneous list.

4.5.2 LCMEQU

Function used to perform a deep-copy of the information contained in an associative table (address IPLIS1) towards another associative table (address IPLIS2). Note that the second associative table (address IPLIS2) is modified but not created by LCMEQU.

CALL LCMEQU(IPLIS1, IPLIS2)

input parameter:		
IPLIS1	TYPE(C_PTR)	address of the existing associative table or of the heterogeneous list (accessed in read-only mode).

output parameter:		
IPLIS2	TYPE(C_PTR)	address of the associative table or of the heterogeneous list, modified by LCMEQU.

4.5.3 LCMEXP

Function used to export (or import) the content of an associative table towards (or from) a sequential file. The sequential file can be in binary or **ascii** format.

The export of information starts from the active directoty. Note that LCMEQU is basically a serialization algorithm based on the contour algorithm.

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

input parameters:		
IPLIST	TYPE(C_PTR)	address of the associative table or of the heterogeneous list to be exported (or imported).
IMPX	INTEGER	print parameter (equal to 0 for no print).
NUNIT	INTEGER	unit number of the sequential file.
IMODE	INTEGER	=1 binary sequential file; =2 ASCII sequential file.
IDIR	INTEGER	=1 to export; =2 to import.

4.6 Using fixed-length character arrays

The following subroutines are implemented using the LCM Fortran API of the preceding sections. They permit the use of fixed-length character arrays. They reproduce an existing capability of the GANLIB4 API.

		type of operation	
		put	get
father	associative table	LCMPTC	LCMGTC
	heterogeneous list	LCMPLC	LCMGLC

4.6.1 LCMGTC

Subroutine used to recover a character array from a block of data stored in an associative table.

CALL LCMGTC(IPLIST,NAMP,LENG,NLIN,HDATA)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the character array to recover. A call to XABORT is performed if the block doesn't exist.
LENG	<i>INTEGER</i>	length of each character variable in the array HDATA.
NLIN	<i>INTEGER</i>	dimension of array HDATA.

output parameter:		
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq NLIN in which the character information is to be copied

4.6.2 LCMPTC

Subroutine used to store a character array into a block of data stored in an associative table. If the block of data already exists, it is updated; otherwise, it is created. This operation cannot be performed in a LCM object open in **read-only** mode.

CALL LCMPTC(IPLIST,NAMP,LENG,NLIN,HDATA)

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the character array to store.
LENG	<i>INTEGER</i>	length of each character variable in the array HDATA.
NLIN	<i>INTEGER</i>	dimension of array HDATA.
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq NLIN from which the character information is recovered.

Example:

```

      USE GANLIB
      ...
      TYPE(C_PTR) :: IPLIST
      PARAMETER (ILONG=5)
      CHARACTER*8 HDATA1(ILONG),HDATA2(ILONG)
*
      CALL LCMOP(IPLIST,'mon_dict',0,1,2)
*
* STORE THE INFORMATION.
      HDATA1(1)='string1'
      HDATA1(2)='string2'
      HDATA1(3)='string3'
      HDATA1(4)='string4'
      HDATA1(5)='string5'
      CALL LCMPTC(IPLIST,'node1',8,ILONG,HDATA1)
*
* RECOVER THE INFORMATION.
      CALL LCMGTC(IPLIST,'node1',8,ILONG,HDATA2)
      DO I=1,ILONG
        PRINT *, 'I=',I, ' RECOVER HDATA2 -->',HDATA2(I), '<--'
      ENDDO
*
      CALL LCMCL(IPLIST,2)

```

4.6.3 LCMGLC

Subroutine used to recover a character array from a block of data stored in an heterogeneous list.

```
CALL LCMGLC(IPLIST, ISET, LENG, NLIN, HDATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
ISET	<i>INTEGER</i>	index of the block in the list. The first element of the list is located at index 1.
LENG	<i>INTEGER</i>	length of each character variable in the array HDATA.
NLIN	<i>INTEGER</i>	dimension of array HDATA.

output parameter:		
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq NLIN in which the character information is to be copied

4.6.4 LCMPLC

Subroutine used to store a character array into a block of data stored in an heterogeneous list. If the block of data already exists, it is updated; otherwise, it is created. This operation cannot be performed in a LCM object open in **read-only** mode.

```
CALL LCMPLC(IPLIST,USET,LENG,NLIN,HDATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
ISSET	<i>INTEGER</i>	index of the block in the list. The first element of the list is located at index 1.
LENG	<i>INTEGER</i>	length of each character variable in the array HDATA.
NLIN	<i>INTEGER</i>	dimension of array HDATA.
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq NLIN from which the character information is recovered.

Example:

```

      USE GANLIB
      ...
      TYPE(C_PTR) :: IPLIST
      PARAMETER (ILONG=5)
      CHARACTER*8 HDATA1(ILONG),HDATA2(ILONG)
*
      CALL LCMOP(IPLIST,'mon_dict',0,1,2)
*
* STORE THE INFORMATION.
      HDATA1(1)='string1'
      HDATA1(2)='string2'
      HDATA1(3)='string3'
      HDATA1(4)='string4'
      HDATA1(5)='string5'
      CALL LCMPLC(IPLIST,1,8,ILONG,HDATA1)
*
* RECOVER THE INFORMATION.
      CALL LCMGLC(IPLIST,1,8,ILONG,HDATA2)
      DO I=1,ILONG
        PRINT *, 'I=',I, ' RECOVER HDATA2 -->',HDATA2(I), '<--'
      ENDDO
*
      CALL LCMCL(IPLIST,2)

```

4.7 Using variable-length character arrays

The following subroutines are implemented using the LCM Fortran API of the preceding sections. They permit the use of variable-length character arrays. They represent a *new capability* of the GANLIB5 API.

		type of operation	
		put	get
father	associative table	LCMPCD	LCMGCD
	heterogeneous list	LCMPCL	LCMGCL

4.7.1 LCMGCD

Function used to recover a character array from a block of data stored in an associative table.


```
CALL LCMGCD(IPLIST,NAMP,ILONG,HDATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the character array to recover. A call to XABORT is performed if the block doesn't exist.
ILONG	<i>INTEGER</i>	number of components in the character array.

output parameter:		
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq ILONG in which the character information is to be copied

4.7.2 LCMPCD

Subroutine used to store a character array into a block of data stored in an associative table. If the block of data already exists, it is updated; otherwise, it is created. This operation cannot be performed in a LCM object open in **read-only** mode.

```
CALL LCMPCD(IPLIST,NAMP,ILONG,HDATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the associative table.
NAMP	<i>CHARACTER*12</i>	name of the character array to store.
ILONG	<i>INTEGER</i>	number of components in the character array.
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq ILONG from which the character information is recovered.

Example:

```

      USE GANLIB
      ...
      TYPE(C_PTR) :: IPLIST
      PARAMETER (ILONG=5)
      CHARACTER*16 HDATA1(ILONG),HDATA2(ILONG)
*
      CALL LCMOP(IPLIST,'mon_dict',0,1,2)
*
* STORE THE INFORMATION.
      HDATA1(1)='string1'
      HDATA1(2)='  string2'
      HDATA1(3)='    string3'
      HDATA1(4)='      string4'
      HDATA1(5)='        string5'
      CALL LCMPCD(IPLIST,'node1',ILONG,HDATA1)
*
* RECOVER THE INFORMATION.
      CALL LCMGCD(IPLIST,'node1',ILONG,HDATA2)
      DO I=1,ILONG
        PRINT *, 'I=',I, ' RECOVER HDATA2 -->',HDATA2(I), '<--'
```

```

        ENDDO
*
        CALL LCMCL(IPLIST,2)

```

4.7.3 LCMGCL

Subroutine used to recover a character array from a block of data stored in an heterogeneous list.

```
CALL LCMGCL(IPLIST,ISET,ILONG,HDATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the heterogeneous list.
ISET	<i>INTEGER</i>	index of the character array in the heterogeneous list. A call to XABORT is performed if the component doesn't exist. The first element of the list is located at index 1.
ILONG	<i>INTEGER</i>	number of components in the character array.

output parameter:		
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq ILONG in which the character information is to be copied

4.7.4 LCMPCCL

Subroutine used to store a character array into a block of data stored in an heterogeneous list. If the block of data already exists, it is updated; otherwise, it is created. This operation cannot be performed in a LCM object open in **read-only** mode.

```
CALL LCMPCCL(IPLIST,ISET,ILONG,HDATA)
```

input parameters:		
IPLIST	<i>TYPE(C_PTR)</i>	address of the heterogeneous list.
ISET	<i>INTEGER</i>	index of the character array in the heterogeneous list. The first element of the list is located at index 1.
ILONG	<i>INTEGER</i>	number of components in the character array .
HDATA	<i>CHARACTER*(*)(*)</i>	character array of dimension \geq ILONG from which the character information is recovered.

Example:

```

        USE GANLIB
        ...
        TYPE(C_PTR) :: IPLIST, JPLIST
        PARAMETER (ILONG=5)
        CHARACTER*16 HDATA1(ILONG),HDATA2(ILONG)
*
        CALL LCMOP(IPLIST,'mon_dict',0,1,2)

```

```

*
* CREATE THE LIST.
      JPLIST=LCMLID(IPLIST,'node2',77)
*
* STORE THE INFORMATION.
      HDATA1(1)='string1'
      HDATA1(2)='  string2'
      HDATA1(3)='    string3'
      HDATA1(4)='      string4'
      HDATA1(5)='        string5'
      CALL LCMPC(L(JPLIST,1,ILONG,HDATA1)
*
* RECOVER THE INFORMATION.
      CALL LCMGCL(JPLIST,1,ILONG,HDATA2)
      DO I=1,ILONG
        PRINT *, 'I=', I, ' RECOVER HDATA2 -->', HDATA2(I), '<--'
      ENDDO
*
      CALL LCMCL(IPLIST,2)

```

4.8 Dynamic allocation of an elementary blocks of data in ANSI C

A function `LCMARA()` and a subroutine `LCMDRD()` are available as wrappers to memory allocator `setara_c` and memory deallocator `rlsara_c` introduced in Sects. 3.8.1 and 3.8.2. `LCLARA()` and `LCMDRD()` offer an alternative to the Fortran-90 `ALLOCATE` and `DEALLOCATE` capabilities for exceptional situations involving *pinning* towards LCM internal data. Use of `LCLARA()` and `LCMDRD()` is 64-bit clean and Fortran-2003 compliant, but its use must be avoided as much as possible. A `setara` address is a `malloc` pointer, as defined in ANSI-C.

4.8.1 LCMARA

`LCMARA()` is a Fortran-2003 wrapper for the ANSI-C function `setara_c` introduced in Sect. 3.8.1. This function perform a memory allocation and returns a `TYPE(C_PTR)` pointer variable.

```
IDATA_PTR=LCMARA(ILONG)
```

input parameter:		
ILONG	INTEGER	length of the data array (in single-precision words).

output parameter:		
IDATA_PTR	TYPE(C_PTR)	setara address of the data array.

4.8.2 LCMDRD

`LCMDRD()` is a Fortran-2003 wrapper for the ANSI-C function `rlsara_c` introduced in Sect. 3.8.2. This subroutine deallocate the memory corresponding to a `TYPE(C_PTR)` pointer variable.

```
CALL LCMDRD(IDATA_PTR)
```

input parameter:		
IDATA_PTR	TYPE(C_PTR)	setara address of the data array.

Example:

```

USE GANLIB
...
TYPE(C_PTR) :: IDATA_PTR
INTEGER, POINTER, DIMENSION(:) :: IDATA
...
IDATA_PTR=LCMARA(50)
CALL C_F_POINTER(IDATA_PTR,IDATA,(/ 50 /))
DO I=1,50
  IDATA(I)=...
ENDDO
...
CALL LCMDRD(IDATA_PTR)

```

4.9 Abnormal termination of the execution

4.9.1 XABORT

Subroutine used to cause the program termination. A message describing the conditions of the termination is printed.

It is important to use this subroutine to abort a program instead of using the **STOP** statement of Fortran. The **XABORT** subroutine can be used to implement *exception treatment* in situations where the application software is driven by a multi-physics system.

If an abnormal termination occurs, the **XABORT** subroutine is called as

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

```
CALL XABORT(HSMG)
```

input parameter:		
HSMG	CHARACTER*(*)	message describing the conditions of the abnormal termination.
value of the function:		
void		

5 The ISO Fortran CLE-2000 API

5.1 Management of Fortran files outside CLE-2000

The KDROPN utility is a general system for managing Fortran files in a software application.

5.1.1 KDROPN

Function used to open a file and allocate its unit number. Allocate a unit number to file name. If unit is already opened, returns its address. Sequential (formatted or not) and direct access (DA) files are permitted.

IFILE=KDROPN(CUNAME,IACTIO,IUTYPE,LRDA)

input parameters:		
CUNAME	<i>CHARACTER*(*)</i>	file name. If <code>cuname=' '</code> , use a default name.
IACTIO	<i>INTEGER</i>	action on file. = 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	<i>INTEGER</i>	file type. = 1: (not used); = 2: sequential unformatted; = 3: sequential formatted; = 4: direct access (DA) unformatted file.
LRDA	<i>INTEGER</i>	number of words in a DA record (used if <code>IUTYPE = 4</code>).

output parameter:		
IFILE	<i>INTEGER</i>	unit number of the allocated file. Equal to the error code if the allocation failed.

5.1.2 KDRCLS

Function used to close a file and release its unit number.

IER=KDRCLS(IFILE,IACTIO)

input parameters:		
IFILE	<i>INTEGER</i>	unit number of the allocated file (as returned by KDROPN).
IACTIO	<i>INTEGER</i>	action on file. = 1: to keep the file; = 2: to delete the file.

output parameter:		
IER	<i>INTEGER</i>	error code. Equal to zero if the close is successful.

5.2 Management of word-addressable (KDI) files outside CLE-2000

The KDIOP utility is a general system for managing word-addressable (KDI) files in a software application.

5.2.1 KDIOP

Function used to open a KDI file and allocate its header.

MY_FILE=KDIOP(CUNAME, IACTIO)

input parameters:		
CUNAME	<i>CHARACTER*(*)</i>	file name.
IACTIO	<i>INTEGER</i>	action on file. = 0: to allocate a new file; = 1: to access and modify an existing file; = 2: to access an existing file in read-only mode.

output parameter:		
MY_FILE	<i>TYPE(C_PTR)</i>	address of the allocated file. Equal to C_NULL_PTR if the allocation failed.

5.2.2 KDIGET

Subroutine used to read a data array from a KDI file at offset IOFSET.

CALL KDIGET(MY_FILE, IDATA, IOFSET, LENGTH)

input parameters:		
MY_FILE	<i>TYPE(C_PTR)</i>	address of the allocated file (as returned by KDIOP).
IOFSET	<i>INTEGER</i>	offset of the information in the KDI file.
LENGTH	<i>INTEGER</i>	length of the array of information, in unit of 32-bit words.

output parameter:		
IDATA	<i>INTEGER</i>	array of information.

5.2.3 KDIPUT

Subroutine used to store a data array in a KDI file at offset IOFSET.

CALL KDIPUT(MY_FILE, IDATA, IOFSET, LENGTH)

input parameters:		
MY_FILE	<i>TYPE(C_PTR)</i>	address of the allocated file (as returned by KDIOP).
IDATA	<i>INTEGER</i>	array of information.
IOFSET	<i>INTEGER</i>	offset of the information in the KDI file.
LENGTH	<i>INTEGER</i>	length of the array of information, in unit of 32-bit words.

5.2.4 KDICL

Function used to close a KDI file.

IER=KDICL(MY_FILE,IACTIO)

input parameters:		
MY_FILE	<i>TYPE(C_PTR)</i>	address of the allocated file (as returned by KDIOP).
IACTIO	<i>INTEGER</i>	action on file. = 1: to keep the file; = 2: to delete the file.
output parameter:		
IER	<i>INTEGER</i>	error code. Equal to zero if the close is successful.

5.3 Management of Fortran and KDI files used as CLE-2000 parameters

CLE-2000 allows a module of the application software to exchange information using LCM objects and files. If the application software is programmed in Fortran, the CLE-2000 driver expects all these parameters to be *TYPE(C_PTR)* variables. The ISO Fortran CLE-2000 API defines a collection of four functions to wrap the KDROP utility in such a way that Fortran files are referred by *TYPE(C_PTR)* variables.

5.3.1 FILOPN

Function used to open a file and allocate its unit number. Allocate a unit number to file name. If unit is already opened, returns its address. Word addressable (KDI), sequential (formatted or not) and direct access (DA) files are permitted. This function is a GANLIB wrapper for the KDROP and KDIOP utilities.

IFILE=FILOPN(CUNAME,IACTIO,IUTYPE,LRDA)

input parameters:		
CUNAME	<i>CHARACTER*(*)</i>	file name. If <i>cuname</i> =' ', use a default name.
IACTIO	<i>INTEGER</i>	action on file. = 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	<i>INTEGER</i>	file type. = 1: KDI word addressable file; = 2: sequential unformatted; = 3: sequential formatted; = 4: direct access (DA) unformatted file.
LRDA	<i>INTEGER</i>	number of words in a DA record (used if <i>IUTYPE</i> = 4).
output parameter:		
IFILE	<i>TYPE(C_PTR)</i>	address of the allocated file. Equal to <i>C_NULL_PTR</i> if the allocation failed.

5.3.2 FILCLS

Function used to close a file and release its unit number. This function is a GANLIB wrapper for the KDRCLS and KDICL utilities.

IER=FILCLS(FILE_PT, IACTIO)

input parameters:		
FILE_PT	TYPE(C_PTR)	address of the allocated file (as returned by FILOPN).
IACTIO	INTEGER	action on file. = 1: to keep the file; = 2: to delete the file.

output parameter:		
IER	INTEGER	error code. Equal to zero if the close is successful.

5.3.3 FILUNIT

Function used to recover the Fortran file unit number

IUNIT=FILUNIT(FILE_PT)

input parameter:		
FILE_PT	TYPE(C_PTR)	address of the allocated file (as returned by FILOPN).

output parameter:		
IUNIT	INTEGER	file unit number. Equal to -1 in case of error.

5.3.4 FILKDI

Function used to recover the address of the KDI file.

KDI_PT=FIKDI(FILE_PT)

input parameter:		
FILE_PT	TYPE(C_PTR)	address of the allocated file (as returned by FILOPN).

output parameter:		
KDI_PT	TYPE(C_PTR)	address of the KDI file. Equal to C_NULL_PTR if case of error.

5.4 The main entry point for CLE-2000

Function `KERNEL` is a Fortran wrapper around function `cle2000_c()` to serve as the main entry point for CLE-2000. Function `KERNEL` is specialized to the case where the application software is executed in stand-alone mode. It is therefore limited to the simple case where a CLE-2000 procedure has no parameters and no in-out CLE-2000 variables. Moreover, the main CLE-2000 procedure is recovered from the standard unit (i.e., from unit 5) and is assumed to have a `.x2m` suffix. This limitation is making sense as no multi-physics system is currently programmed in Fortran.

5.4.1 KERNEL

The general specification of function `KERNEL` is


```
IER=KERNEL(DUMMOD,IPRINT)
```

input parameters:		
DUMMOD	<i>EXTERNAL</i>	external C-interoperable Fortran-2003 function responsible for dispatching the execution among calculation modules.
IPRINT	<i>INTEGER</i>	print parameter (set to zero for no print).

output parameter:		
IER	<i>INTEGER</i>	error code. Equal to zero if the execution of KERNEL is successful.

5.4.2 DUMMOD

Function KERNEL has one of its arguments that is a developer-defined external function. Function DUMMOD is a C-interoperable Fortran-2003 function responsible for dispatching the execution among calculation modules. An instance of function DUMMOD is implemented for each Fortran application software using the GANLIB.

A stand-alone GANLIB application can be set by using the following implementation of GANMOD

```
!
!-----
!
!Purpose:
! Dispatch to a calculation module in GANLIB. ANSI-C interoperable.
!
!Copyright:
! Copyright (C) 2009 Ecole Polytechnique de Montreal
! This library is free software; you can redistribute it and/or
! modify it under the terms of the GNU Lesser General Public
! License as published by the Free Software Foundation; either
! version 2.1 of the License, or (at your option) any later version.
!
!Author(s): A. Hebert
!
!-----
!
integer(c_int) function GANMOD(cmodul, nentry, hentry, ientry, jentry, &
                             kentry, hparam_c) bind(c)
!
    use GANLIB
    implicit none
!----
!  subroutine arguments
!----
    character(kind=c_char), dimension(*) :: cmodul
    integer(c_int), value :: nentry
    character(kind=c_char), dimension(13,*) :: hentry
    integer(c_int), dimension(nentry) :: ientry, jentry
    type(c_ptr), dimension(nentry) :: kentry
```

```

    character(kind=c_char), dimension(73,*) :: hparam_c
!----
!  local variables
!----
    integer :: i, ier
    character :: hmodul*12, hsmg*131, hparam*72
    character(len=12), allocatable :: hentry_f(:)
    type(c_ptr) :: my_file
    integer, external :: GANDRV
!
    allocate(hentry_f(nentry))
    call STRFIL(hmodul, cmodul)
    do i=1,nentry
        call STRFIL(hentry_f(i), hentry(1,i))
        if(ientry(i) >= 3) then
!           open a Fortran file.
            call STRFIL(hparam, hparam_c(1,i))
            my_file=FILOPN(hparam,jentry(i),ientry(i)-1,0)
            if(.not.c_associated(my_file)) then
                write(hsmg,'(29hGANMOD: unable to open file ',a12,2h''.)') hentry_f(i)
                call XABORT(hsmg)
            endif
            kentry(i)=my_file
        endif
    enddo
!  -----
    GANMOD=GANDRV(hmodul,nentry,hentry_f,ientry,jentry,kentry)
!  -----
    do i=1,nentry
        if(ientry(i) >= 3) then
!           close a Fortran file.
            ier=FILCLS(kentry(i),1)
            if(ier < 0) then
                write(hsmg,'(30hGANMOD: unable to close file ',a12,2h''.)') hentry_f(i)
                call XABORT(hsmg)
            endif
        endif
    enddo
    deallocate(hentry_f)
    flush(6)
    return
end function GANMOD

```

with function GANDRV implemented as

```

integer function GANDRV(hmodul,nentry,hentry,ientry,jentry,kentry)
!
!-----
!
!Purpose:
! standard utility operator driver for Ganlib.
!
!Copyright:
! Copyright (C) 2002 Ecole Polytechnique de Montreal

```

```

! This library is free software; you can redistribute it and/or
! modify it under the terms of the GNU Lesser General Public
! License as published by the Free Software Foundation; either
! version 2.1 of the License, or (at your option) any later version
!
!Author(s): A. Hebert
!
!Parameters: input/output
! hmodul  name of the operator.
! nentry  number of LCM objects or files used by the operator.
! hentry  name of each LCM object or file.
! ientry  type of each LCM object or file:
!         =1 LCM memory object; =2 XSM file; =3 sequential binary file;
!         =4 sequential ascii file.
! jentry  access of each LCM object or file:
!         =0 the LCM object or file is created;
!         =1 the LCM object or file is open for modifications;
!         =2 the LCM object or file is open in read-only mode.
! kentry  LCM object address or file unit number.
!
!Parameters: output
! kdrstd  completion flag (=0: operator hmodul exists; =1: does not exists).
!
!-----
!
!----
!  subroutine arguments
!----
  use GANLIB
  integer nentry
  character hmodul*(*),hentry(nentry)*12
  integer ientry(nentry),jentry(nentry)
  type(c_ptr) kentry(nentry)
!
  GANDRV=0
  if(hmodul == 'EQU:') then
!    standard equality module.
    call DRVEQU(nentry,hentry,ientry,jentry,kentry)
  else if(hmodul == 'GREP:') then
!    standard grep module.
    call DRVGRP(nentry,hentry,ientry,jentry,kentry)
  else if(hmodul == 'UTL:') then
!    standard LCM/XSM utility module.
    call DRVUTL(nentry,hentry,ientry,jentry,kentry)
  else if(hmodul == 'ADD:') then
!    standard addition module.
    call DRVADD(nentry,hentry,ientry,jentry,kentry)
  else if(hmodul == 'MPX:') then
!    standard multiplication module.
    call DRVMPX(nentry,hentry,ientry,jentry,kentry)
  else if(hmodul == 'STAT:') then
!    standard compare module.
    call DRVSTA(nentry,hentry,ientry,jentry,kentry)

```

```

      else if(hmodul == 'BACKUP:') then
!       standard backup module.
          call DRVBACK(nentry,hentry,ientry,jentry,kentry)
      else if(hmodul == 'RECOVER:') then
!       standard recovery module.
          call DRVREC(nentry,hentry,ientry,jentry,kentry)
      else if(hmodul == 'FIND0:') then
!       standard module to find zero of a continuous function.
          call DRV000(nentry,hentry,ientry,jentry,kentry)
      else if(hmodul == 'MSTR:') then
!       manage user-defined structures.
          call MSTR(nentry,hentry,ientry,jentry,kentry)
      else if(hmodul == 'MODUL1:') then
!       user-defined module.
          call DRVMO1(nentry,hentry,ientry,jentry,kentry)
      else if(hmodul == 'ABORT:') then
!       requested abort.
          call XABORT('GANDRV: requested abort.')
      else
          GANDRV=1
      endif
      return
end function GANDRV

```

5.5 The free-format input reader

Subroutines REDOPN, REDGET, REDPUT and REDCLS are Fortran wrappers around ANSI C functions `redopn_c()`, `redget_c()`, `redput_c()` and `redcls_c()`. Only REDGET and REDPUT are expected to be used in an application software.

5.5.1 REDOPN

Subroutine REDOPN is called to open the input reader. The general specification of function REDOPN is

```
CALL REDOPN(IINP1,IOUT1,NREC)
```

input parameters:		
IINP1	<i>TYPE(C_PTR)</i>	KDI object containing the CLE-2000 input data.
IOUT1	<i>INTEGER</i>	unit number of the sequential ASCII file used to write execution messages. Can be set to 6 for standard output.
NREC	<i>INTEGER</i>	record index where reading occurs. Can be set to zero at first call. Set to the value returned by REDCLS at subsequent calls.

5.5.2 REDGET

Subroutine REDGET is called within modules of the application software to recover the module-specific input data. The general specification of function REDGET is

```
CALL REDGET(ITYP,NITMA,FLOTT,TEXT,DFLOT)
```

output parameters:		
ITYP	<i>INTEGER</i>	type of the CLE-2000 variable. A negative value indicates that the variable is to be computed by the application software and returned towards CLE-2000 using a call to <code>redput_c</code> . = ± 1 : integer type; = ± 2 : real (single precision) type; = ± 3 : string type; = ± 4 : double precision type; = ± 5 : logical type.
NITMA	<i>INTEGER</i>	integer input value when ITYP = 1 or = 5; number of characters when ITYP = 3.
FLOTT	<i>REAL</i>	real input value when ITYP = 2.
TEXT	<i>CHARACTER*(*)</i>	character string input value when ITYP = 3.
DFLOT	<i>DOUBLE PRECISION</i>	double precision input value when ITYP = 4.

5.5.3 REDPUT

Subroutine REDPUT is called within modules of the application software to make information computed by the module available as CLE-2000 variables to the CLE-2000 procedure. The application software must first call REDGET and obtain a negative value of ITYP. A call to REDPUT is next performed with its first parameter set to $-ITYP$ (now, a positive value) and with the corresponding value of the parameter. The general specification of function REDPUT is

```
CALL REDPUT(ITYP,NITMA,FLOTT,TEXT,DFLOT)
```

input parameters:		
ITYP	<i>INTEGER</i>	type of the CLE-2000 variable. = 1: integer type; = 2: real (single precision) type; = 3: string type; = 4: double precision type; = 5: logical type.
NITMA	<i>INTEGER</i>	integer output value when ITYP = 1 or = 5; number of characters when ITYP = 3.
FLOTT	<i>REAL</i>	real output value when ITYP = 2.
TEXT	<i>CHARACTER*(*)</i>	character string output value when ITYP = 3.
DFLOT	<i>DOUBLE PRECISION</i>	double precision output value when ITYP = 4.

5.5.4 REDCLS

Subroutine REDCLS is called to close the input reader. The general specification of function REDCLS is

```
CALL REDCLS(IINP1,IOUT1,NREC)
```

output parameters:		
IINP1	<i>TYPE(C_PTR)</i>	KDI object containing the CLE-2000 input data.
IOUT1	<i>INTEGER</i>	unit number of the sequential ASCII file used to write execution messages.
NREC	<i>INTEGER</i>	record index where reading occurs.

References

- [1] R. Roy, *The CLE-2000 Tool-Box*, Report IGE-163, Institut de génie nucléaire, École Polytechnique de Montréal, Montréal, Québec (1999).
- [2] A. Hébert, *LCM – Guide du programmeur*, Rapport IGE-296, Institut de génie nucléaire, École Polytechnique de Montréal, Montréal, Québec (2002).
- [3] B. W. Kernighan and D. M. Ritchie, *The C programming language, second edition*, Prentice Hall, Englewood Cliffs, New Jersey (1988).
- [4] M. Metcalf, J. Reid and M. Cohen, *Fortran 95/2003 explained*, Oxford University Press, Oxford, U. K. (2004).
- [5] A. Hébert and R. Roy, “A Programmer’s Guide for the GAN Generalized Driver, FORTRAN-77 version”, Report IGE-158, Institut de Génie Nucléaire, École Polytechnique de Montréal, December 1994.
- [6] A. Hébert, “Coarse-Grain Parallelism Using Remote Method Invocation”, paper submitted at the *International Conference on Supercomputing in Nuclear Applications*, Paris, France, September 22 – 24 (2003).

Index

- cle2000.c, 3
- clecls, 13
- clecst, 16
- clelib, 14
- clemod.c, 9
- clenode, 13
- cleopn, 12
- clepop, 13
- clepos, 14
- clepush, 13

- DUMMOD, 69
- dummod, 3

- FILCLS, 67
- FILKDI, 68
- FILOPN, 67
- FILUNIT, 68

- KDICL, 67
- KDIGET, 66
- KDIOP, 66
- KDIPUT, 66
- KDRCLS, 65
- KDROPN, 65
- KERNEL, 68

- LCMARA, 63
- LCMCL, 42
- lcml.c, 20
- LCMDEL, 48
- lcmdel.c, 27
- LCMDID, 52
- lcmdid.c, 30
- LCMDIL, 54
- lcmdil.c, 32
- LCMDRD, 63
- LCMEQU, 57
- lcmequ.c, 35
- LCMEXP, 57
- lcexp.c, 36
- LCMGCD, 60
- lcmgcd.c, 36
- LCMGCL, 62
- lcmgcl.c, 38
- LCMGDL, 48
- lcmgdl.c, 27
- LCMGET, 45
- lcmget.c, 24
- LCMGID, 55
- lcmgid.c, 34
- LCMGIL, 56
- lcmgil.c, 34
- LCMGLC, 59
- LCMGPD, 46
- lcmgpd.c, 26
- LCMGPL, 50
- lcmgpl.c, 29
- LCMGTC, 58
- LCMINF, 43
- lcminf.c, 22
- LCMLEL, 44
- lcml.c, 23
- LCMLLEN, 43
- lcmlen.c, 21
- LCMLIB, 57
- lcmlib.c, 35
- LCMLID, 52
- lcmlid.c, 31
- LCMLIL, 53
- lcmlil.c, 32
- LCMNXT, 44
- lcmnxt.c, 23
- LCMOP, 41
- lcmap.c, 20
- LCMPCD, 61
- lcmpcd.c, 37
- LCMPCL, 62
- lcmpcl.c, 38
- LCMPDL, 49
- lcmpdl.c, 28
- LCMPLC, 59
- LCMPPD, 47
- lcmppd.c, 26
- LCMPPL, 51
- lcmppl.c, 29
- LCMPTC, 58
- LCMPUT, 46
- lcmp.c, 25
- LCMSIX, 56
- lcmsix.c, 35
- LCMVAL, 42
- lcmval.c, 21
- lifo, 11

- REDCLS, 74
- redcls.c, 16
- REDGET, 73
- redget.c, 15
- REDOPN, 72
- redopn.c, 14
- REDPUT, 73
- redput.c, 15

rlsara.c, 40

setara.c, 39

strcut.c, 19

strfil.c, 19