Use of JEVEUX

Summary:

It is a question here of indicating some concepts of operation of the manager of memory JEVEUX in order to specify the use of the routines “user”, to indicate the routines the best appropriate ones to certain actions and to announce the difficulties of use. One presents here the rules of use in italic in each paragraph.

Contents

1 Use of the bases....3
2 Access by name....3
3 Access to the segments of values....3
3.1 Access by pointer....4
3.2 Access by address....4
4 Initialization of the values....5
5 Release of the segments of values and concept of mark....5
6 Detection of crushings report....7
7 To increase a vector....7
8 The recopy of the objects....7
9 Routines working on groups of objects....8
10 How to effectively exploit the contiguous and numbered collections?....8
10.1 How to reach quickly the elements of a numbered and contiguous collection?....9
10.2 How to create a numbered and contiguous collection quickly ?....9

1 Use of the bases

The simple objects can be created by the routines `JECREO` and `WKVECT`, collections by the routine `JECREC`.

`WKVECT` allows to connect the three calls `JECREO`, `JEECRA` and `JEVEUO` for an object of vector kind.

The basic concept makes it possible to associate the various objects with a file saved or not at the end of the work. The objects to be preserved at the end of the work will be created on the basis `TOTAL` associated with the class `G`. This base makes it possible to preserve the structures of data and to carry out continuations.

The objects of work will be created on the basis `BIRD` associated with the class `V`. This base is destroyed at the end of work (it is even cleaned between each order). By convention, the characters will be used `&&` at the beginning of the name of any object associated with this base.

It is pointed out that it is not possible to have objects of identical name on different bases. The routine `I….` do not have among their arguments the name of the class and the scan for the noun is carried out in the whole of the repertoires associated with the various open bases.

.

base `TOTAL`	name of class `G`	preserved objects
base `BIRD`	name of class `V`	temporary objects

Note:

It is also possible to dynamically create vectors of entireties, realities, complexes, character strings,… without passing by jeveux. "Macros" AS_ALLOCATE and AS_DEALLOCATE should then be used.

The differences between a jeveux vector FORTRAN and object are:

- A jeveux object can be discharged on disc (then reloaded in memory). A vector FORTRAN exists only in memory.
- A jeveux object has a name which one can transmit in the form of character string.
- It is faster to create a vector FORTRAN than a jeveux object.
- A jeveux object can be part of a structure of data named (sd_maillage, sd_modele,…).

2 Access by name

The access to the objects managed by `JEVEUX` is carried out using the name. One uses a function of coding which provides starting from the name and of various parameters a key of access (an entirety), this key then allows to reach the various attributes. The access by name is relatively expensive (decoding of characters, management of collision, etc…) also one preserves in a variable the last name (of simple object, collection and object of collection) and the identifier (obtained starting from the key) associated, to avoid a new call to the function of coding.

Note:

It is thus recommended to carry out all the requests for the same object `JEVEUX` in a sequential way in order to profit from this possibility.

3 Access to the segments of values

One reaches the contents of an object JEVEUX thanks to the routine jeveuo (or wkvect).

There are two types of different accesses for each one of these routines:

- an access by "pointer"
- an access by "address in a variable of reference"

The first access (allowed thanks to FORTRAN 2003) is from now on privileged

3.1 Access by pointer

When one wants to reach contents of an object `JEVEUX`, one declares a variable local of type "pointer" on a vector of the good type.
The call to `jeveuo` associate the pointer with the zone memory occupied by the object (this one is then brought back in memory so necessary).
One can then handle the object like a simple vector FORTRAN.

Example: Access in reading to the coordinates of the nodes of the grid:

```
! declaration of the local variable:
real (kind=8), to point:: coordo (:) => no one ()


…
cal jeveuo (my '.COORDO    .VALE', 'It, vr=coordo)
n1=size (coordo)! Access to the length of the object (=3*nbno)
C ino=1, nbno
  x=coordo (3* (ino-1) +1)
  Y=coordo (3* (ino-1) +2)
  …
enddo
```

Notice important:
At the time of the declaration of the local variable (here: `coordo`), it is forced to initialize to point it to "zero" (`=> no one ()`). If not, the pointer is in an unspecified state.

3.2 Access by address

Routines `jeveuo` and `wkvect` can return to the user a relative address in one of the tables ZR , ZI , ZC , or ZK , (one will note thereafter $Z?$ one of these variables FORTRAN). This address is valid as long as there no was release.

The concept of access in writing or reading makes it possible to avoid systematic unloading on disc of the segment of values and limit thus the number of the inputs/outputs on disc. The objects reached in reading will not be saved on disc at the time of the release. The call to `WKVECT` carry out a request in writing.

A segment of values could be reached in writing then released, the manager then preserves it in memory and differs his unloading on disc at the time of a forthcoming research of place. A new access in reading returns the address of the segment déchargeable, a modification of the contents can thus take place unbeknownst to the user if this last affects the contents of the table $Z?$ with the address indicated.

<u>*Rule of use:*</u>

The user, when it carries out an access in reading, should not modify the contents of the table $Z?$ with the address provided at the time of the request and must avoid passing it in argument of a subroutine of which it would not have the whole control.

The call to the routine `jeveuo` return, the address of object JEVEUX relative to a variable $Z?$ of the same type as the object `JEVEUX` (this address is measured in the length of the type).

The standardized commun run must appear in any unit of program carrying out this kind of call.
Since version NEW11.2.2 this commun run is inserted in the sources by the instruction:

#include `"jeveux.h"`

who will automatically come to substitute the following instructions for compilation:

```
INTEGER ZI
COMMON/IVARJE/ZI (1)
INTEGER*4 ZI4
COMMON/I4VAJE/ZI 4(1)
REAL*8 ZR
COMMON/RVARJE/ZR (1)
COMPLEX*16 ZC
COMMON/CVARJE/ZC (1)
LOGICAL ZL
COMMON/LVARJE/ZL (1)
CHARACTER*8  ZK8
CHARACTER*16 ZK16
CHARACTER*24 ZK24
CHARACTER*32 ZK32
CHARACTER*80 ZK80
COMMON/KVARJE/ZK 8(1), ZK 16(1), ZK 24(1), ZK 32(1), ZK 80(1)
```

Note:

Not to modify the noun of the variables of the commun run of reference.

The access to a segment of values is carried out in the following way: if $JTAB$ indicate the address returned by the routine JEVEUO for an object of vector kind and type I, $KTAB$ that for an object of the type C (complex):

.

$ZI(JTAB)$	is the first value of a vector of entireties,
$ZI(JTAB+I-1)$	is I $^{\text{éme}}$ value of a vector of entireties,
$ZC(KTAB+I-1)$	is I $^{\text{éme}}$ value of a vector of complexes.

Variables (JTAB) likely to contain an address JEVEUX and used in argument of the routines in the form ZI (JTAB), ZR (JTAB), etc, must always be initialized with value 1. Thus, if this address is not the result of a call to JEVEUO or WKVECT, one will point on a valid value for the first element within the meaning of the access to the memory. ZI (1), ZR (1) and ZC (1) are initialized with a value being able to cause an error or an operation leading to Not.

4 Initialization of the values

At the time of the call to jeveuo or with wkvect the manager of memory carries out a search for place in main memory. If the object does not have an image on disc (i.e. at the time of its first access in writing), the segment of value is initialized according to the type of the object: $0.$ for realities, $(0.,0.)$ for the complexes, 0 for the entireties, ' ' (white) for the characters.

Rule of use:

It is useless to carry out a loop of initialization before the first use of the segment of values.

5 Release of the segments of values and concept of mark

If nothing is made (not call with `JELIBE`), the segments of values brought back in memory remain there and that can lead quickly to its saturation. On another side, if one releases an object without taking precautions (a mark), one is likely to return invalid the address of an object requested upstream of the programming.

What is recommended:

- one uses the marks to release the objects (routines `JEMARQ` / `JEDEMA`).
- very few objects explicitly are released (largest) what makes it possible to make sure that the releases are not dangerous (good knowledge of the use of these objects);

6 Detection of crushings report

JEVEUX allocate dynamically each zone memory associated with the segments with values at the time of the first request. The various segments of values associated with the objects are framed by 4 entireties in front of and 4 behind. These entireties make it possible to store the state and the statute, the identifier and the class as well as a shift to manage the types of length higher than the unit of addressing.

The crushing of the entireties located around the segment causes the loss of the identity of the object associated with the segment with values. Such a crushing is generally detected at the time of the writing on disc of the contents of the object and not at the time of the overflow. It results in one of the following error messages:

```
 POSSIBLE <S> <JJLIRS> <ECRASEMENT UPSTREAM ADRESSE> nnnn
```

In this case one crushed one of the entireties located in front of the segment of values.

```
 POSSIBLE <S> <JJLIRS> <ECRASEMENT DOWNSTREAM ADRESSE> nnnn
```

In this case one crushed one of the entireties located behind the segment of values.

Rule of use:

The developer has the routine then JXVERI *to instrument its code.*

This routine checks the integrity of the values on both sides of segment of values and can announce the point of rupture. It detects in more one incursion out of the zone licit memory. It is possible to implement at less expenses in the order BEGINNING or CONTINUATION the call to this subroutine before each order, which makes it possible to determine which order carries out crushing. The developer will then be able, by instrumenting the routines associated with the order, to proceed by dichotomy to determine the erroneous routine or instructions.

Crushing can also be less important and to affect only one word in front of or behind the segment of values, it is the case when one makes an error of index in the table $Z?$. The contents of the use or the statute of the segment of value are then affected, this information can be obtained by consulting the result of the impressions of the distribution in memory by the routine JEIMPM.

7 To increase a vector

Rule of use:

The user has the routine juveca *to increase a simple object of vector kind.*

It is a surcouche written starting from the routines user JEVEUX. It builds a temporary object and destroys the original after recopy.
After having increased an object, it is necessary to ask again a pointer on this object (jeveuo).
These various operations can be rather expensive, it is thus preferable to minimize the number of calls to juveca for the same object.
A current strategy is to double the size of object when it is noted that its size is insufficient (rather than to increase it progressively.

8 The recopy of the objects

It is possible to recopy the objects JEVEUX on the same basis or from one base to another. The recopy of the simple objects does not pose a particular problem, on the other hand it is more delicate to handle collections. A collection can be pressed on an external repertoire of names or an external pointer length. These simple objects must be created and partly managed independently (by example their destruction must be explicit). Their name can thus be without relationship with the name of the collection.

Two cases are possible:

- the recopy is carried out on different bases: the external pointers will be duplicated and will become internal pointers with the collection,
- the recopy is carried out on the same basis: the external pointers can be preserved or it are duplicated and become interns.

If the receptacle is already existing, it is destroyed before recopy.

__Rules of use:__

To use `jedupo`

9 Routines working on groups of objects

The organization of the structures of data of Aster rests mainly on the names of the objects. One handles within the code of the "concepts" built starting from a name provided by the user like result of the orders. It thus appeared convenient to be able to handle group of objects by providing a under-chain of characters, which is required in the names of all the objects present in the repertoires.

Routines `jedetc` and `jedupc` apply to lists of objects. They make it possible, in the order, to release, destroy and duplicate the objects of these lists.

These routines offer more flexibility to the developer to manage the objects (structures of data) but they are less effective than the routines "into hard" `detrsd` and `copisd`.

__Rule of use:__

Not to use the routines `jedetc` and `jedupc`. To use in the place `detrsd` and `copisd`.

10 How to effectively exploit the contiguous and numbered collections?

It happens sometimes that one creates collections very having one a large number of elements. For example, the connectivity of the grid is a collection which can have several hundreds of thousands of elements (an element by mesh).
When one must "buckle" on the elements of these collections, the use of the jeveux functions (`jeveuo, will jelira`,...) on each object of the collection can become very expensive.

When the collection "contiguous" and is numbered, we explain in the two following paragraphs, how to avoid the use of the jeveux functions in the loop on the elements.

This more effective use is directly related to the presence of a system object: the "pointer cumulated lengths". This object exists only for the contiguous collections.

10.1 How to reach quickly the elements of a numbered and contiguous collection?

It is supposed here that one wants to reach in a loop many elements of a numbered contiguous collection (here, all elements).
The collection is called `COLLEC`, it has `nbobj` elements.

10.1.1 "Classical" access:

```
C iobj=1, nbobj
    ! recovery length of the object of collection (dimobj):
    cal will jelira (jexnum (COLLEC, iobj), 'LONMAX', ival=dimobj)
    ! recovery of the address the object of collection (jcollec):
    cal jeveuo (jexnum (COLLEC, iobj), 'It, jcollec)
    cal xxxxx (Zr (jcollec),…)
enddo
```

10.1.2 "Optimized" access:

```
cal jeveuo (COLLEC, 'It, jcollec1)
cal jeveuo (jexatr (COLLEC, 'LONCUM'), 'It, jlc_collec)

C iobj=1, nbobj
    ! calculation length of the object of collection (dimobj):
    dimobj=zi (jlc_collec+iobj) - zi (jlc_collec+iobj-1)
    ! calculation of the address the object of collection (jcollec):
    jcollec=jcollec1-1+zi (jlc_collec+iobj-1)
    cal xxxxx (Zr (jcollec),…)
enddo
```

10.2 How to create a numbered and contiguous collection quickly ?

It is supposed here that one wants to create a collection (`COLLEC`) contiguous numbered having `nbobj` elements. The elements of the collection have a priori different lengths.
It is also supposed that one calculated as a preliminary the cumulated length (`ltot`) of ALL the elements of the collection.

10.2.1 "Classical" creation :

```
cal jecrec (COLLEC, 'G V I', 'NAKED', 'CONTIG', 'VARIABLE', nbobj)
cal will jeecra (COLLEC, 'LONT', ival=ltot)
C iobj=1, nbobj
    cal jecroc (jexnum (COLLEC, iobj))
    dimobj =…
    cal will jeecra (jexnum (COLLEC, iobj), 'LONMAX', ival=dimobj)
    cal jeveuo (jexnum (COLLEC, iobj), 'E', jcollec)
    ! filling of the iobj object:
    zi (jcollec) =…
enddo
```

10.2.2 “Optimized” creation :

```
cal jecrec (COLLEC, ‘G V I’, ‘NAKED’, ‘CONTIG’, ‘VARIABLE’, nbobj)
cal will jeecra (COLLEC, ‘LONT’, ival=ltot)
cal jeveuo (COLLEC, ‘E’, jcollec1)
cal jeveuo (jexatr (COLLEC, ‘LONCUM’), ‘E’, jlc_collec)
zi (jlc_collec) =1
C iobj=1, nbobj
    dimobj =…
    ! address of the iobj object (jcollec):
    decal=zi (jlc_collec-1+iobj)
    jcollec=jcollec1-1+decal
    ! filling of the iobj object:
    zi (jcollec) =…
    ! equivalent of will jelira/LONMAX:
    zi (jlc_collec-1+iobj+1) =decal+dimobj
enddo
! To replace ALL “jecroc”:
cal will jeecra (COLLEC, ‘NUTIOC’, nbobj)
```

Notice important:

The writing of `NUTIOC` is a a little expensive operation. It is thus very important to make this operation only once (at the end of the construction of the collection) and not in the loop on the elements.