DYNOSOR, a set of subroutines for dynamic memory organization in FORTRAN programs

M. Huybrechts

Institut Interuniversitaire des Sciences Nucléaires

Brussels

1.- Introduction

1.1 What's missing in FORTRAN ?

FORTRAN is certainly the computer language which is most widely used for scientific calculations.

It suffers however from serious drawbacks, perhaps the most important of these being the absence of dynamic memory allocation. Dimensions of arrays have to be specified at compilation time while the real needs depend often on the data read and are therefore known only at execution time.

This leaves the FORTRAN user with only a choice between two bad solutions : either to overestimate the dimensions of arrays, which often uses unnecessary computer memory, or to adjust these dimensions to each particular case, which wastes computer time for repeated compilations. Furthermore, both solutions involve risks of errors which also wastes computer time as well as user's time.

1.2 DYNOSOR adds dynamic allocation to FORTRAN

The aim of the DYNOSOR system is to allow the FORTRAN user to overcome these difficulties without the effort of learning another computer language.

The interface between the DYNOSOR system and the user's program consists of Fortran subroutines used simply by CALL statements without any modification of the Fortran compiler.

To help the user to remember them easily, the names of these subroutines are obtained by concatenation of two-character codes.

1.3 Specifications can be given at execution time

The information needed by the DYNOSOR system for memory organization can be given at execution time, which allows adaptation to the particular data read by each program.

1.4 DYNOSOR corrects wrong specifications

Some of these specifications can, however, still be unknown before the execution of the program starts. E.g. if one wants to store, among the data read, only those having a specific property, their number can be unknown before all the data have been read in.

Therefore, the DYNOSOR system considers the specifications provided by the user only as rough estimates of the real needs. Wrong specifications don't produce fatal errors. They only reduce the efficiency of the program.

1.5 DYNOSOR helps the user to correct his mistakes

When wrong specifications are corrected, the user is informed about it. This allows the user to give better values for future runs and so to improve the efficiency of his program.

1.6 DYNOSOR and other dynamic allocation methods

Many Fortran users have already developped their own dynamic allocation method adapted to their specific needs. HYDRA is a well known example in the field of high energy physics (see Duby 1971).

The DYNOSOR system has been developped, on the contrary, without relation to a specific problem. It should be able to organize any set of data which are to be referenced by names or by linkage to a tree structure.

Special care has been taken to make its use easy by
a) the use of mnemonic subroutine names.
b) the acceptance of wrong specifications.
c) the information displayed to the user.

An approach similar to DYNOSOR has been developped some years ago by J.M. SAKODA with his DYSTAL system. It seems however that DYSTAL is less flexible than DYNOSOR, as only the last created array can be deleted (see SAKODA 1968).

2.- Data organization

2.1 Banks

The data are stored by DYNOSOR into the computer central memory. They are grouped in blocks of consecutive words of a Fortran array (called VDYN) located in the blank common.

The length of each data block is defined at execution time and can be modified by the user or by the DYNOSOR system itself. Each data block can be linked to another one which is considered as its continuation.

A BANK consists of one or more of these data blocks, linked together. It can therefore be defined as a string of chained data blocks. In the present description, however, we consider only banks consisting of one single data block. For the general case, see the DYNOSOR user's guide.

A bank is defined by its NAME, which is transmitted as an argument to all bank handling subroutines. A special name is used for a special type of bank, called a VERTEX BANK and which is linked to a tree structure (see 9.).

2.2 Location of banks

As a result of garbage collection, the data blocks can be moved during the execution of a program. Therefore, the addresses of data blocks must be obtained again, from the DYNOSOR system, before each use. However, indirect addresses (i.e. addresses of words containing the addresses of data blocks) are not affected by garbage collection and can thus be used at any time. This allows a much faster retrieval of data blocks than the use of bank names.

2.3 Bank names

A bank name is a string of alphanumeric characters. The number of characters is computer dependent as it is related to the word length. The argument "bkname" transmitted to the DYNOSOR subroutines is either a Hollerith constant (nH...) or a variable whose value has been previously defined as a Hollerith constant (NAME=nH...).

3.- Subroutine names

Most of the subroutines of the DYNOSOR system which can be called directly by the user have names obtained by concatenation of two-character codes.

The first code always represents an operator. It is followed by one or two codes representing operands.

The list of these codes is given in the table below, as well as the subroutine names obtained with them and the list of their arguments.

Operator	1st operand	2nd operand	
AJ (adjust)	BK (bank)	--	AJBK(bkname)
CP (copy)	AR (array)	BK	CPARBK(nwds,$\underline{\text{arname}}$,bkname)
	BK	AR	CPBKAR(nwds,bkname,iwd,$\underline{\text{arname}}$)
	BK	BK	CPBKBK(nwds,bkname1,iwd1,bkname2,iwd2)
DF (define)	BK	--	DFBK(bkname,nwds)
	VX (vertex)	--	DFVX(vxname,nbranch)
DL (delete)	BK	--	DLBK(bkname)
	WD (word)	BK	DLWDBK(nwds,bkname,ifrstwd)
DS (describe)	BK	--	DSBK(bkname,$\underline{\text{nbls}}$,$\underline{\text{nwds}}$)
IN (insert)	WD	BK	INWDBK(nwds,bkname,ilstwd,$\underline{\text{lfrstwd}}$)
LC (locate)	BK	--	LCBK(bkname,$\underline{\text{lbl}}$,$\underline{\text{nwds}}$)
MD (modify)	BK	--	MDBK(bkname,nwds)
PR (print)	BK	--	PRBK(bkname,iwdin,iwdfin,ncoln,fmt)
RN (rename)	BK	--	RNBK(bkname1,bkname2)
TR (transfer)	AR	BK	TRARBK(nwds,$\underline{\text{arname}}$,bkname)
	BK	BK	TRBKBK(nwds,bkname1,iwd1,bkname2)
UN (unify)	BK	--	UNBK(bkname)

Underlined arguments must be replaced by variables.
Nonunderlined arguments can be replaced by variables or by constants.

4.- Definition of a bank

The instruction

```
                CALL DFBK(bkname,nwds)
```

defines a new BANK having the name "bkname" and consisting of one single data block containing "nwds" words. The value of "nwds" can be defined at execution time and is only an estimate of the bank length really needed.

The bank length can be modified later, either by the user, with calls to the subroutines AJBK or MDBK (see 7.1 and 7.3), or by the DYNOSOR system itself.

5.- Filling of a bank

To avoid confusion with the definition of a new bank, which only reserves a set of central memory words for future use, we use the word "filling" for the definition of the contents of the words of a previously defined bank.

Filling can be done either by a copy or a transfer of data.

By a transfer, we mean the writing of data just after the data previously written by a copy (CP) or a transfer (TR) instruction. Thus a copy can override data previously copied or transferred. A transfer cannot.

5.1 Filling a bank from a Fortran array

The instruction

```
                CALL CPARBK(nwds,arname,bkname,iwd)
```

copies "nwds" words from Fortran array "arname" into bank "bkname".
The first word copied is the first word of the array and it is written into word

"iwd" of the bank. To start the copy at word "i" of the array, argument "arname(i)" should be used instead of "arname".

```
CALL TRARBK(nwds,arname,bkname)
```

copies also "nwds" words from array "arname", but the first word is written just after the last word previously copied into the bank by a CP... or TR... subroutine.

Example

Definition and filling of a bank, whose length is estimated as 1000 words. The number of data is a multiple of 8 and an end-of-file is used to indicate the end of the data.

```
      DIMENSION CARD(8)
      .....
      CALL DFBK(5HYBANK,1000)             (defines bank YBANK)
   10 READ (IUNIT,1001) CARD
      IF(EOF(IUNIT).NE.0) GO TO 15
      CALL TRARBK(8,CARD,5HYBANK)
      GO TO 10
   15 CALL AJBK(5HYBANK)                  (AJ=adjust, see 7.1)
      .....
 1001 FORMAT(8F10.4)
      .....
```

The program also works when the number of data read is larger than the estimated length of 1000 words, but the efficiency is better when the estimated length is larger than the real length.
CPARBK can be used instead of TRARBK by substituting the following instructions

```
      IWD=1
   10 READ(IUNIT,1001) CARD
      IF(EOF(IUNIT).NE.0) GO TO 15
      CALL CPARBK(8,CARD,5HYBANK,IWD)
      IWD=IWD+8
      GO TO 10
```

When the number of values to be read is not a multiple of 8, the last value can be recognized by the fact it is followed by a special value as 99999.9999.
To remove the values read in excess, the following instructions can be added to the instructions of the example above.

```
      DO 1 I=1,8
      IF(CARD(I).EQ.99999.9999) GO TO 20
    1 CONTINUE
      GO TO 25
   20 NXCESS=9-I
      CALL DSBK(5HYBANK,NBLS,NWDS)                  (DS=describe)
      NWDS=NWDS-NXCESS
      CALL MDBK(5HYBANK,NWDS)                       (MD=modify)
   25 .....
```

alternatively, the last three instructions can be replaced by

```
      CALL MDBK(5HYBANK,-NXCESS)
```

which reduces the bank length by the value NXCESS (see 7.3).

5.2 Filling a bank from another bank

```
CALL CPBKBK (nwds,bkname1,iwd1,bkname2,iwd2)
```

copies "nwds" words, the first word copied being word "iwd1" from bank "bkname1", which is copied into word "iwd2" of bank "bkname2".

```
CALL TRBKBK (nwds,bkname1,iwd1,bkname2)
```

performs the same operation as above, but the first word is copied just after the last word previously written into the bank by a CP.... or a TR.... subroutine.

Copy with a step

By inserting a CALL STEPDYN (istep) before a call to a CP.... or TR.... subroutine, the copy is performed with a step equal to "istep". The argument "nwds" represents the number of words effectively copied and not the number of words among which the copied words are chosen.

The value of the step is reset to 1 after each call to a CP.... or TR.... subroutine.

6.- Use of a bank

A bank can be used in different ways :

6.1 It can be copied into another bank by CPBKBK or TRBKBK.
This can be used to merge several banks into one single bank or to extract data from a bank, when the copy is done with a step.

6.2 It can be copied into a Fortran array (Don't forget to define it by a DIMENSION statement!).
This is done by a

```
CALL CPBKAR(nwds,bkname,iwd,arname)
```

which copies "nwds" words, starting at word "iwd" from bank "bkname" into Fortran array "arname".

Caution, no error message is printed by DYNOSOR if the number of words copied is larger than the dimension of the array.

The copy into a Fortran array can sometimes reduce computer time. E.g. a two-dimensional array can be used for matrix manipulations, the results being stored back later into a bank.

6.3 Data blocks can be localized by a

```
CALL LCBK(bkname,lbl,nwds)
```

This instruction returns the value of "lbl", which is the address of the first word of the data block of bank "bkname". This word however contains only information used by DYNOSOR and the data are stored in words lbl + 1 to lbl + nwds of array VDYN. To have access to the data, it is necessary to insert the instruction

```
COMMON//IERRDYN,VDYN(1)
```

in each subroutine which calls LCBK.

The use of LCBK allows also the filling of a bank without use of CP.... or TR.... subroutines.

7.- Some other operations on banks

7.1 Adjustement of bank length

When a bank has been filled by CP..BK or TR..BK subroutines, its length can be adjusted to the number of words copied by these subroutines by a

```
CALL AJBK(bkname)
```

7.2 Deletion of a bank

A bank can be deleted by a

```
CALL DLBK(bkname)
```

The name "bkname" can be reused later for another bank.

To avoid unnecessary garbage collection, a call to DLBK does not lead to the immediate deletion of the bank, but only to a marking for future deletion, when garbage collection occurs.

7.3 Modification of bank length

The bank length can be modified by the user with the instruction

```
CALL MDBK(bkname,newnwds)
```

"newnwds" being the new length of the bank. Even when "newnwds" is larger than "nwds" the bank will still consists of only one data block. Increase of bank length can use a lot of computer time, when a bank is inserted between other banks. In this case, data have to be copied from one place to another in central memory. Therefore, it is generally better to overestimate the bank length than to underestimate it. A call with a negative value for "newnwds" subtracts |newnwds| to the bank length.

7.4 Renaming a bank can be done with a

```
CALL RNBK(bkname,newbknm)
```

which gives the new name "newbknm" to the bank previously defined by the name "bkname". The old name can be used again for the definition of a new bank.

7.5 Insertion of words into a bank can be done by a

```
CALL INWDBK(nwds,bkname,ilstwd,ifrstwd)
```

which inserts "nwds" words after word "ilstwd" of bank "bkname". The location of the first inserted word is returned via "lfrstwd". The contents of the inserted words are not defined by INWDBK. This can be done by Fortran instructions (instruction COMMON//IERRDYN,VDYN(1) must then be present) or by CP..BK or TR..BK subroutines.

Insertion of words splits a bank into at least three data blocks. A bank having only one data block can be obtained by

```
CALL UNBK(bkname)
```

When several insertions are done, computer time can be saved by calling UNBK only at the very end of the process.

7.6 Deletion of words of a bank can be done by a

```
CALL DLWDBK(nwds,bkname,ifrstwd)
```

which deletes "nwds" words, starting at word "ifrstwd" of bank "bkname". The bank length is adjusted.

7.7 Use of "nwds" = 0

Subroutines CPBK.. and TRBK.. can be called with an argument "nwds" equal to zero. In that case the whole bank is considered. In the same way, a call to PRBK with "iwdfin" = 0 prints the contents of the bank, starting with word "iwdin", until the last word.

8.- Initialization

The statement CALL INITDYN must be executed before any other call to a subroutine of the DYNOSOR system.

This instruction initializes the parameters of the DYNOSOR system by using default values which are convenient for most usual cases. However, other values can be assigned to these parameters (see DYNOSOR user's guide).

9.- VERTEX BANKS and tree structure

A Vertex bank is not referenced by a name, but is linked to a vertex of a tree structure.

A simple example of the use of a tree structure is the storage of multi-dimensional arrays, with any number of indices. To each of these indices corresponds a vertex bank, and as the bank lengths can be different, space can be saved (e.g. the storage of a very big symmetric matrix).

A tree structure has to be defined before any vertex bank can be. It must be described step by step, starting from the root of the tree (vertex of zero level). The definition can be done horizontally, all the vertices of a given level being defined before the next level is considered, or vertically, each defined branch being then continued until its end before another branch is defined, or by a combination of the two methods.

A vertex is defined by the statement

```
          CALL DFVX(2H**,nbranch)
```

"nbranch" being the number of branches which can be attached to the newly defined vertex and the identificator of the new vertex being transmitted via the Fortran array IVXDYN, located in common bloc VXDYN.

The statement

```
          COMMON/VXDYN/IVXDYN(n)
```

must therefore be present in the calling subroutine.

n is an integer constant related to the maximum number of levels of the tree, which is equal to n - 1.

Let us assume that it is required to define a new vertex of level L having the identifier $(I_1,I_2,\ldots I_{L-1},I_L)$ and linked to the vertex $(I_1,I_2,\ldots I_{L-1})$ previously defined with a number of branches at least equal to I_L.

The following statements are needed therefor

```
          IVXDYN(1)=I1
          IVXDYN(2)=I2
          ..........
          IVXDYN(L)=IL
          IVXDYN(L+1)=0
          CALL DFVX(2H**,nbranch)
```

IVXDYN(L+1) has to be set equal to zero. This defines the level L of the new vertex.

10.- Garbage collection

Garbage collection is completely under the control of the DYNOSOR system and is performed when needed, without any intervention of the user. However, the user can specify the criteria which force the system to perform garbage collection, by defining properly the appropriate parameters at the initialization stage.

In the DYNOSOR system, garbage collection is performed in a specially simple way. This is due to the fact that the dynamic store is divided into two distinct areas.

The first one, the A-area, is located in the lower part of array VDYN and contains only information about memory organization.

The second one, the D-area, is located in the upper part of array VDYN and contains the data, grouped into data blocks. Only the first word of each of these data blocks contains information about memory organization(mainly a backward pointer to the A-area).

No garbage collection occurs into the A-area. Therefore, updating of pointers is very easy. When a block is moved in the D-area, only the contents of its indirect address has to be changed, the indirect address itself remaining unchanged.

Nevertheless, the length of the A-area can be increased (and also decreased). This does not need a shifting of the whole D-area. Only the blocks located near the A-area jump over other blocks to be copied,either into free holes or at the very end of the D-area.

11.- Portability

The DYNOSOR system exists in two versions :

1) CDC Version, series 6000.
   This version has been tested on a CDC 6500 at the Université Libre de Bruxelles and should work without any modification on the CDC series 7000 and CYBER 70.
   In this version, it is made use of the 60 bit length of the words and whenever possible, 3 integers are stored into one word.
   To improve the speed, some subroutines are written in COMPASS.
2) Portable version
   In this version, only one value is stored into each word. All the subroutines are written in Fortran, but some instructions are perhaps not compatible with ANSI Fortran and some variables have 7-character names, which is not accepted by most computers.

12.- Complete description of DYNOSOR

This paper gives only the basic features of the DYNOSOR system. A more sophisticated use allows the user, once he is familiarized with the system,to improve greatly the speed of programs using it. The complete description can be found in the DYNOSOR user's' guide which can be obtained by writing to the author at following address : Dr Marcel HUYBRECHTS
Université Libre de Bruxelles
Campus de la Plaine CP 228
Boulevard du Triomphe
B-1050 Bruxelles (Belgium)

References

Duby 1971 Proceedings of the 1970 CERN computing and processing School. CERN Yellow report 71-6 pp393-398.

Sakoda 1968 DYSTAL : Dynamic Storage Allocation Language in FORTRAN, in "Symbol manipulation languages and techniques", BOBROW J.G. Ed., North Holland 1968.