To be submitted to
Computer Physics
Communications

ISTITUTO NAZIONALE DI FISICA NUCLEARE
Laboratori Nazionali di Frascati

LNF-78/6(P)
31 Gennaio 1978

R. Bernabei, S. d'Angelo and A. Marini:

FILE MANAGEMENT FORTRAN ROUTINES.

Servizio Documentazione
dei Laboratori Nazionali di Frascati
Cas. Postale 13 - Frascati (Roma)

INFN - Laboratori Nazionali di Frascati
Servizio Documentazione

LNF-78/6(P)
31 Gennaio 1978

R. Bernabei[(x)], S. d'Angelo[(x)] and A. Marini: FILE MANAGEMENT FORTRAN ROUTINES.

PROGRAM SUMMARY

Title of program : FMFR - File Management Fortran Routines

Catalogue number : -

Program obtainable from : -

Computer : IBM 370/135 ; Installation : Laboratori Nazionali di Frascati

Operating system : DOS-POWER/VS (release 31), OS/VS1 (release 6)

Programming language used : FORTRAN IV

High speed storage required: 26.5 min, 57.4 tip, 108 max, K bytes

No. of bits in a byte : 8

Overlay structure : none

No. of magnetic tape required : none

Other peripheral used : line printer, direct access device

No. of cards in combined program and test deck : 897

Card punching code : EBCDIC

CPC library subprograms used: none

Keywords : direct access device, direct access fortran statements, file, record, physical record, logical record, hashing, scatter storage, links, pointers, linked chains

---

(x) Istituto di Fisica dell'Università, and INFN - Sezione di Roma.

Nature of problem

To use efficiently the direct access statements: DEFINE FILE, READ, WRITE, FIND; supplied by the FORTRAN IV, the user needs to keep memory of where each record is located in the file. Therefore, to update the file or simply to retrieve the informations it may be a hard job for different users.

Method of solution

The file is subdivided into logical elements which are randomly accessed by means of a table of contents, recorded in the same file. Each element is supposed to be built up with records that will be processed sequencially or will have a structure of linked chains. Storing or retrieving the informations from an element all one needs to know is its name.

Restrictions

Some slight modifications are unavoidable for an efficient use on files with very small capacities (less about 10000 records).

Unusual features of the program

Direct-access I/0 statements.

References

(1) - D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, 1973), pp. 508-518.

1. - Introduction.

The FORTRAN language, by means of the statements DEFINE FILE, WRITE, READ, FIND, allows the scientific programmer to access directly to any record within a file defined on a direct access device : drum, disk, etc.

As a matter of practice some difficulties arise when retrieving records or updating the file. In fact one needs to know the address of each physical record, say its number relative to the beginning of the file. A common approach consists of thinking the records as grouped in "elements", built from logically homogeneus records that, usually will be processed sequencially, while the elements themselves can be processed in any order. If a unique name identifies each element with in the file, it sufficies to keep some sort of table of contents. It will give the address of the first record of each element and allows to have access to the entire file.

We present in this paper a package of FORTRAN routines that allows a such management in a very simple way. By means of them, in fact, one can insert a new element into a file ; read the text of each element, in whatever order ; delete elements ; recover space become available after deletion ; obtain a list of the content of the entire file or a list of any element, in its convenient format.

These routines, in the version described below, are currently in use on the IBM 370/135 computer at Laboratori Nazionali di Frascati, and these have been used under the DOS/POWER-VS operating systems to create, on IBM 2314 disks, a data base of high energy measurenets from an experiment performed at ADONE storage ring. Looking forward to next experiments and computer's configuration changes, they have been designed in such a way to easily fit different devices and also different software features. So now the routines run also, without modification, under OS/VS1 operating system with IBM 3330 disks.

2. - File and directory organization.

The file is formatted in records, 240 eight-bit bytes long, the first 200 of which are used to keep a directory of the elements loaded into. The remaining records are used to store elements, as shown in Fig. 1.

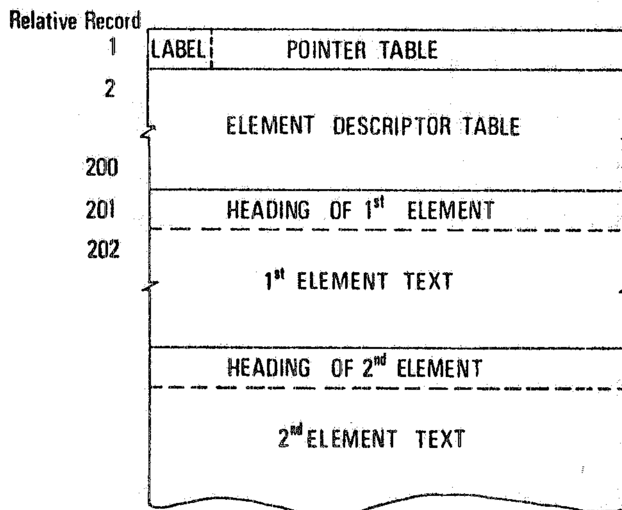

FIG. 1 - File format.

The directory structure reflects the "scatter storage" technique (1) employed. Shortly it consists in the following: the text of a new element is written onto the file, starting from the first empty record following the last element loaded, as shown in Fig. 2. The first record number of this new element, say its text address relative to the beginning of the file, is written, with element name and length, into

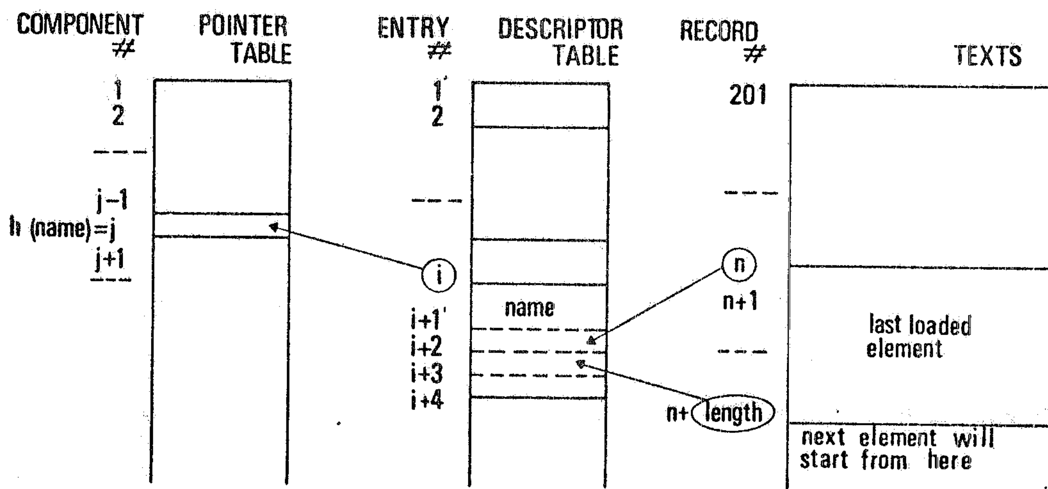

FIG. 2 - Schematic of scatter-storage technique.

some consecutive entries of an "element descriptor table" (IET). The number of the first entry is then recorded in a component of an array (IPT) that is used as "pointer table" to retrieve the element. That component of IPT in which is pointed each element, is calculated by means of an "hashing function" of the name :

$$h(\text{name}) = 1 + \left| \text{name}-1 + \text{name}-2 \right| \bmod 103 \qquad (1)$$

where name - 1 and name - 2 represents the first and last four bytes of the element name ; $|a|$ represents the absolute value of a ; mod b indicates the remainder of the division by b.

This function will give only 103 different values, so we must foresee that elements with different names can give the same value for h. In fact, on each entry of the element descriptor table we leave room to accomodate the pointer to another element which has a different name but equal hash code.

Finally we must consider the case in which a new element replaces an old one or, simply, we want to delete some element. This is accomplished by changing the sign of its length : a negative length will means that the element is physically presents in the file, but no longer available. In Fig. 3 it is shown in detail the format of IET : 8 bytes are reserved for the element name ; 4 bytes each for the text address, text length and link to another element with the same hash code. As shown in Fig. 4

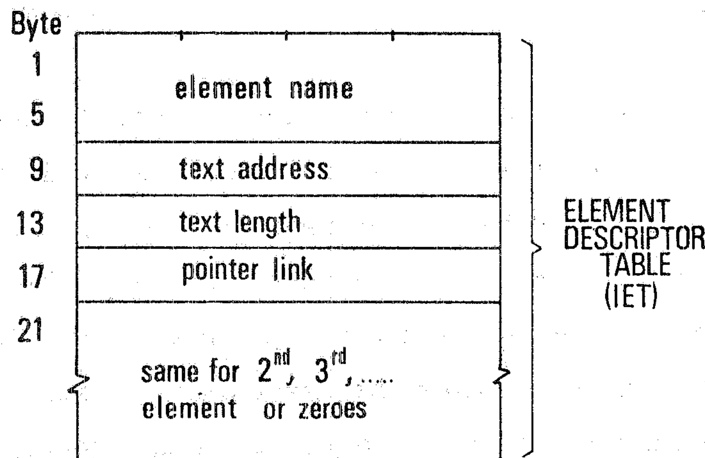

FIG. 3 - Element descriptor format.

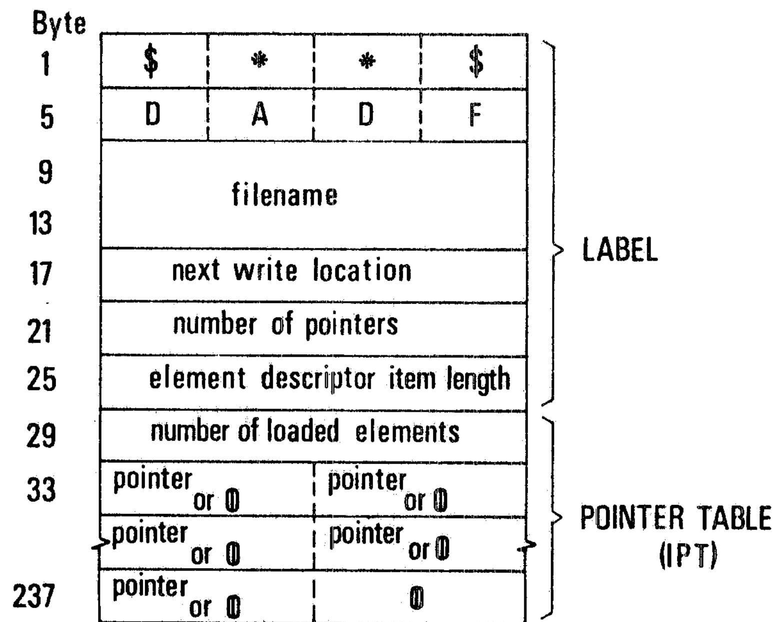

FIG. 4 - Label and Pointer table format.

first 32 bytes are reserved to keep a label of the file. It consists of 8 characters to tag the file, 8 to accomodate the file name, 4 for each of the following: the next empty record, the number of pointers, the descriptor of item length and the total number of loaded elements.

3. - The block data.

All significant variables, with their initial values, are defined in the block data routine, subdivided in four labelled common blocks. They are:

a) The common block /DISKS/ in which are defined the associated variables for the various files defined on direct access devices, numbered up to a maximum of 14, excluding units 5, 6, 7, which are reserved to reader, printer and card puncher, and excluding unit 1, for which it is impossible to share the value in an array. Note that IBM FORTRAN does not allow the associated variable to be an indexed one and a such common block is a trick to make it possible.

b) The common block /SPCØ1/ defines space in which the tag (DADFI)

and directory (LABELI, IPTI, IETI) can be temporary stored from files which have to be read.

c) The common block /SPCØ2/ defines space in which tag (DADFO) and directory (LABELO, IPTO, IETO) are read, updated and from which they will be rewritten into the file, when a new element has been inserted; everywhere, whitin the subroutines, variables of both common blocks are named: DADF, LABEL, IPT, IET because no ambiguity is possible.

d) The common block /PRMØ1/ defines the parameters: MXELEM, the maximum number of elements that can be inserted in the file: 1188; MXETBL, the maximum length of the element descriptor table: 1188 x 5 = 5940; NPOINS, the number of different pointers allowable: 103; NFRREC, the first empty record in the file: 201, when the file is empty; ITMLEN, the number of entries in IET, for each element: 5; RED and REDS, logical variable and array to avoid to read and/or write files that have not been properly initializated.

4. - How to initialize.

Before of any reference to the file on symbolic unit number n, we need to code, somewhere in the program, the following statements:

```
COMMON/DISKS/ IVR (n - 1), IVn
DEFINE FILE n (nrecs, 240, L, IVn)          (2)
```

where nrecs is an integer constant specifying the total number of records on the file.

Furthermore the file must be created, preferably in a separate job, using the call:

```
CALL SFOCRE (FILNAM, IUN)                    (3)
```

where FILNAM is an array, 8 bytes long, containing the name that, in future, will refer to the file; IUN is an integer variable or constant which contains the value n of the symbolic unit.

We stress that the statement (3) must be executed only once during the life of the file, because it initializes non existing files but also erases existing ones.

5. - How to write.

To insert a new element in the file, besides the (2), the file must be enabled to accept new data. It is accomplished by the call:

```
CALL SFOOPN (FILNAM,IUN,ELNAME,FMT)                (4)
```

where FILNAM and IUN have the same meaning as in (3), ELNAME is an array, 8 bytes long, where is stored the name of the new element, and FMT is an array, 240 bytes long, that contains the format specifications under which the element can be dumped. This format will be written as first record of the element.

For each record we want to insert, we need a call:

```
CALL SFOOTP (IMAGE,LENGTH)                         (5)
```

where IMAGE is a buffer, variable in length, that contains the informations to be stored, namely the logical record; LENGTH is an integer variable or constant that for each logical record specifies the number of image words as if these were 4 bytes long:

$$\text{length} = \left\lceil \frac{\#\text{ of image's bytes}}{4} \right\rceil$$

where $\lceil a \rceil$ means the minimum integer greater or equal to a.

When all the images have been written, it is necessary to close the file in order to write back the updated directory, so we need the statement:

```
CALL SFOCLO .                                      (6)
```

6. - How to read. -

In order to search the element to be read, we need the statement:

CALL SFIOPN(FILNAM, IUN, ELNAME, FTM, &nn) (7)

where FILNAM, IUN, ELNAME must be specified like in (4); FMT has the same meaning as in (4), but now is filled from the first record of the element. The program will jump to the statement number nn if the element specified cannot be found.

After element has been found, we can retrieve one logical record at the time, by the statement:

CALL SFINPT(IUN, IMAGE, LENGTH, &nn) (8)

where IUN, IMAGE and LENGTH have same meaning as in (4), but now LENGTH and IMAGE will be filled with informations from the file. nn represents the instruction number at which program will pass control when the end of the element is reached.

7. - How to read a chain of records. -

As "chain of records" we mean records that have embedded, in some fixed location, the number of the one which must follow . All numbers are referred to the beginning of the element.

To be able to retrieve records in this predefined order, we need to open the element by the statement:

CALL SCROPN(FILNAM, IUN, ELNAME, FMT, &nn) (9)

where arguments have same meaning than in (7).

In addition we need the statement:

CALL SCRFIX(OFFSET, TAG, EOC) (10)

whose argument are integer variables or constants.
OFFSET value defines the first record to be read; TAG value defines in

what couple of byte is located the link to the next record, EOC specifies what value of the link is used to indicate the end of the chain. Subsequen tly records can be get, one at time, using statement:

CALL SCRNPT(IMAGE, LENGTH, &nn, &mm) (11)

where IMAGE, LENGTH and nn have same meaning as in (8), instead the program will jump to the statement number mm, when the last record of the chain is reached before the end of the element.

8. - Some utility. -

In order to get a table of contents or a dump of some element, it is possible to use the subroutine LXST, that is called by the statement:

CALL LXST(FILNAM, IUN, ELNAME, IFLAG) (12)

where FILNAM, IUN and ELNAME have same meaning than before, whi le IFLAG is an integer variable or constant that specifies either that we need the table of contents (IFLAG=0) or the number of bytes for each word of the element (IFLAG=1, 2, 3; 2 ** IFLAG=number of bytes/words). The format used for the dump is the one stowed as first record of the ele ment. This format must not specify more than 120 characters/line, columns 121 through 130 being used to number each logical record. Notice that the subroutine LXST alters the content of the common bloch/SPCØ1/, because this is used as buffer to read images.

Insertion of a new element deletes the one with same name. However it is also possible to delete some element by the statement:

CALL DXL(FILNAM, IUN, ELNAME) (13)

The presence of deleted elements waste space in the file and in the directory also. It is possible to recover this space by the statement:

CALL PXCK(FILNAME, IUN) (14)

We outline that the execution of this routine is time expensive and if may seem like an endless loop. Furthermore halting the execution results in loosing all the informations of the directory that is, necessary, re-built only at the end of process. Therefore it is recommended to use this statement with caution and keeping, somewhere, a backup copy of the file.

This subroutine also alters the contents of the common block/ SPCØ1/which, however, would become meaningless all the same.

9. - Internal subroutines. -

The subroutines SFINPT and SCROPN call the subroutine FXNDE, that searches through the common block/SPCØ1/ the element name: elname.

The subroutines SFOCLO and PXCK call the subroutines UCOPY and UZERO that are present in almost all computer lybraries but that have been added for sake of completess and, also because very often they differ sligthly in the argument number or order.

The subroutine SCROPN calls also the subroutine IHALVE, that placed in a 4-bytes word a couple of bytes aligned to any 2-bytes word.

APPENDIX A - Diagnostics. -

The following messages can be issued under various conditions of error. These are illustrated with the entries that determine the message, the probable error and the program response.(# stands for any digit or leading blank, @ stands for any alphanumeric character or trailing blank).

a) execution terminates

1) ** ** @@@@@@@@ DOESN'T CORRESPOND TO FILE NAME.
issued from: SFIOPN, SFOOPN, SCROPN, LXST, DXLT, PXCK;
the argument filnam in these subroutines was specified incorrectly so that the eigth character string shown in the message does not correspond to file name.

2) ** ** FILE # # # HAS A BAD LABEL.
issued from: SFIOPN, SFOOPN, SCROPN, LXST, DXLT, PXCK;
the first eigth characters of the label are not the standard ones. The file was not created with the FMFR package or, more likely, the block data has not been properly included in the collection of the load module.

3) ** ** INDEX TABLE OVERFLOW. EL. NOT LOADED.
issued from SFOOPN;
there is no more space in the directory of the file indicated. If there is enough room, create another file on the same device.

4) ** ** INPUT ELEMENT ON UNIT # # # NEED TO BE OPENED.
issued from: SFINPT, SCRNPT;
the statements SFINPT, SCRFIX or SCRNPT have been executed before of the corresponding statements SFIOPN or SCROPN; or attempt was made to perform SCRFIX or SCRNPT after any end of chain has been encountered.

5) ** ** OUTPUT ELEMENT ON UNIT # # # NEEDS TO BE OPENED.
issued from: SFOOTP, SFOCLO;
the statement SFOOTP or SFOCLO have been not proceded by execution of the statement SFOOPN.

b) execution continues, request is ignored

6) ** ** @@@@@@@@ NOT FOUND.
issued from: LXST, DXLT;
the element indicated by the string cannot be located in file. It does not exist or it is deleted.

APPENDIX B - Suggestions. -

- The FORTRAN statement for the file definition:

DEFINE FILE N (LENGTH, LREC, FMT, IV) (15)

is not a standard one. In fact the sintax is the same for the most of the compilers, e.g. IBM FORTRAN G or H, UNIVAC FORTRAN V, PDP 11 FORTRAN IV plus; but the actual arguments are of different type and also can have slightly different meaning.

We found that the IBM FORTRAN compiler is the less flexible one, in the sense that all argument, but IV, must be integer constants and not variables, and IV must be a variable, but not an indexed one. This implies that a program could be written only for files that have the same length and that will be defined on the same symbolic unit N. Luckily the parameter "length" in the (15) is always ignored under DOS/VS operating system. Under OS/VS1 it is important only the first time the file is referenced, when the corresponding job control statement assumes: DISP = NEW, otherwise the parameter is ignored. So, in order to make the program as general as possible, we can suggest to replace the (2), with the statement:

CALL DEFINE (IUN)

where DEFINE is a subroutine structured like in Fig. 5. Note that, in this subroutine, all references to the files specify the maximum length admissible for the physical device (in our example 266, 665, say about a whole 3330 disk). Obviously the first reference to the file must be made in a separate job, using the (2) and the actual value for the parameter length so that, in this step, one can also "create" the file.

- For historical reasons only, the subroutines waste 100 records of the file, in fact the directory occupies 24,000 bytes so that it can be accomodated in only 100 records, if written without format. When debbugging was a job, we choose to write the directory in an exadecimal format and, because space was not a problem, this feature is yet alive. If you need those 100 records you must remove the format specifications in reading and writing the directory - the only ones you can find on direct access READ, WRITE statements - and you must replace with 101 the value for NFRREC in the block data.

If 100 records are important, perhaps you are using the package on small disks of a computer smaller than IBM 370/135. In this case also the space in the computer memory may be a problem. The subroutines themselves, but PXCK, do not occupy so much space (see Appendix C) the most being by common blocks/SPCØ1/and/SPCØ2/, but it is unlike that one pretend to load up to 1188 elements on small disks. So we can suggest to vary, according your own needs, only the dimensions of arrays IETI, IETO, IET and the values of MXELEM, MXETBL, NFRREC. Values

PROGRAM

```
      SUBROUTINE DEFINE(IUN)
      COMMON /DISKS/ IVR(1),IV2,IV3,IV4,IV5,IV6,IV7,IV8,
     *               IV9,IV10,IV11,IV12,IV13,IV14
      IF(IUN.LE.1) GO TO 90
      IF(IUN.GT.14) GO TO 90
      GO TO (90,2,3,4,90,90,90,8,9,10,11,12,13,14),IUN
    2 CONTINUE
      DEFINE FILE 2(266665,240,L,IV2)
      RETURN
    3 CONTINUE
      DEFINE FILE 3(266665,240,L,IV3)
      RETURN
 ..........
 ..........
   14 CONTINUE
      DEFINE FILE 14(266665,240,L,IV14)
      RETURN
   90 CONTINUE
      PRINT 9000,IUN
      CALL EXIT
 9000 FORMAT('0 ** **',I2,' IS INVALID UNIT.')
      END
```

FIG. 5 - Example of a possible subroutine DEFINE.

being not critical - we use the same on 2314 and 3330 disks - one can choose the following, for example in the case of disk capacity of 5 M-bytes: dimension of IETI, IETO, IET: 900, MXELEM = 180, MEXTBL= 900, NFRREC = 33 (17 if you do not format the directory). This results in a length of 3848 bytes for the common blocks/SPCØ1/and/SPCØ2/.

APPENDIX C. -

In the following we summarize the names, the entry points and the total length of each control section.

CONTROL SECTION NAME	ENTRY POINTS	LENGTH bytes decimal
DISKS	-	56
SPCØ1	-	24008
SPCØ2	-	24008
PRMØ1	-	80
SFOCLO	SFOCRE	2530
	SFOOPN	
	SFOOTP	
	SFOCLO	
SFINPT	SFIOPN	1434
	SFINPT	
SCROPN	SCROPN	1822
	SCRFIX	
	SCRNPT	
LXST	LXST	2236
DXLT	DXLT	1084
PXCK	PXCK	51722
FXNDE	FXNDE	676
IHALVE	IHALVE	328
UCOPY	UCOPY	552
	UZERO	