A General File Management System for the CDC 6600

User's Manual
Version II

MKC.
[illegible]

by
Douglas A. Kellogg*

Technical Report 73001 January, 1973

University Computer Center
University of Minnesota
Minneapolis, Minnesota
55455

*presently with the Hybrid Computer Laboratory, University of Minnesota.

CONTENTS

I. INTRODUCTION

The system of subprograms operating on the 6600 as a file management system (henceforth known as FMS) is designed for using a disk based file and a hashing algorithm for random access of information. It is made to be operated in as little as 30K of core and use a minimum number of disk accesses. It gives the user the options of storing data in ECS - for a savings in time - and of using overlays - for a savings in space. There are a number of system parameters which are to be varied by the user for the greatest efficiency for his particular file. These parameters will be detailed later, along with reasons for selecting one value over another.

File Structure

Each main program using the FMS must declare a labeled common named /FMS/ with a certain minimum length, that length depending on the values selected for the various parameters mentioned above. (See figure 1) The smallest unit of the file is a page, which is composed of from 1 to 28 64-word PRU's. Thus, the length of a page is 64 times the number of PRU's per page, which is user defined. The next larger unit is the slot, which is used for transferring data from the page buffer, which the FMS subprograms use, and the disk (or ECS if that is used for intermediate storage). The I/O is done on the basis of a slot. Each slot consists of m consecutive pages plus a three-word header. The slots make up the bulk of the labeled common area; there are n slots per labeled common. Both m and n are user-defined, and are to be varied for greatest efficiency depending on the structure of the file. Unless ECS storage is used, this is all there is to the FMS labeled common. However, for big files, huge slots in ECS may be used. Data in ECS may be accessed almost as rapidly as data in central memory, and much faster than data on the disk. The structure of the ECS

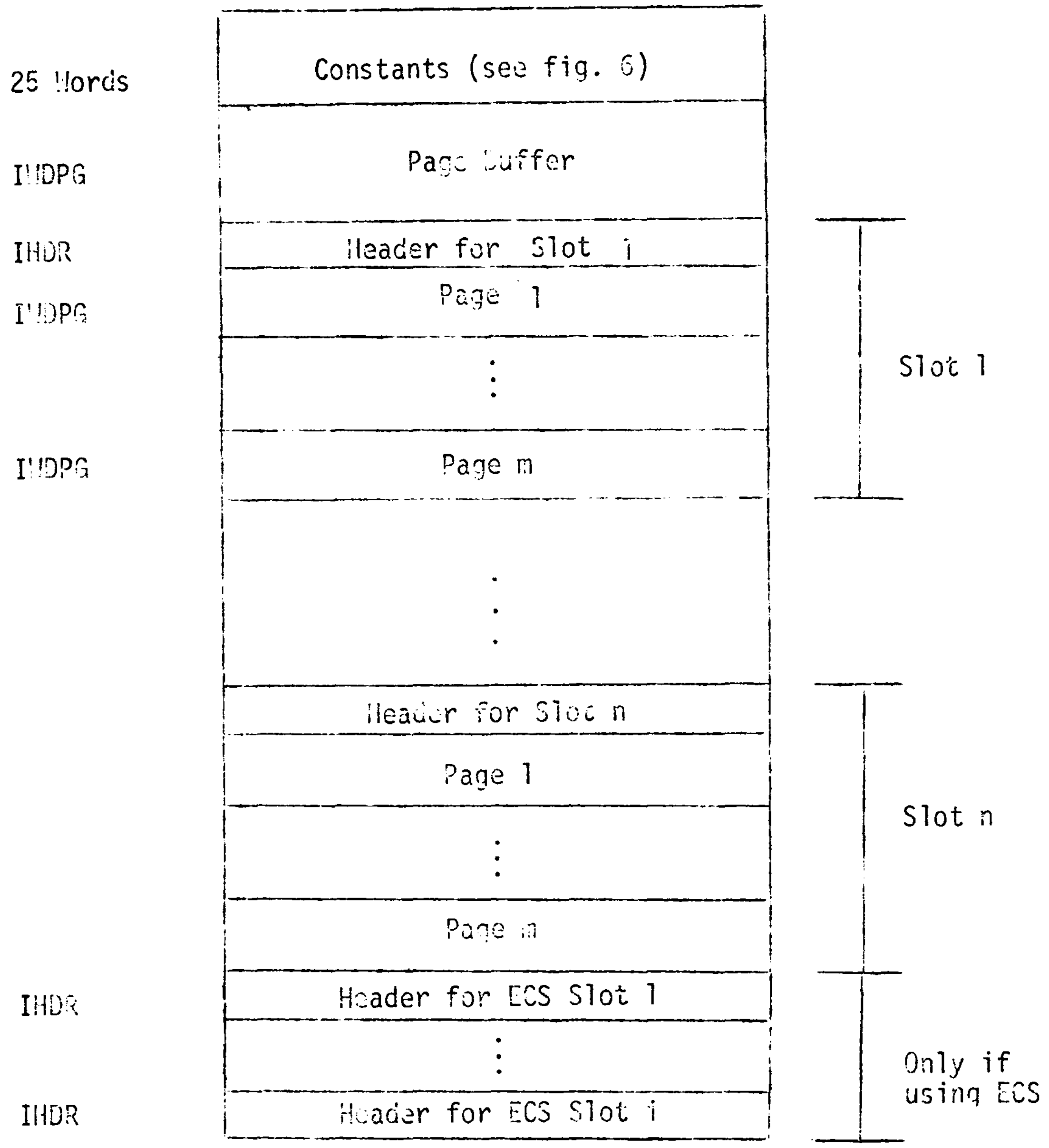


Figure 1. Labeled Common /FMS/

slots is similar to that of the in-core slots: there are i slots and j pages per slot. At the end of the labeled common area there are 3*i header words for the ECS slots. Since, at the present time, the 6600 cannot transfer data directly from disk to ECS, an area of core must be set aside to transfer from disk to core to ECS. One of the slots (the one to be filled by the page desired) is used for the transfer. Thus, when using ECS, the slot size also determines the number of disk I/O operations needed to fill up an ECS slot.

The FMS allows the structuring of data through the use of pointers. In discussing this structure, it is common to use genealogical terminology to describe the relationships. Thus, if we have a pointer set from a first entity to a second we refer to the first as the 'parent' and the second as the 'child'. Entities which have the same parent, and thus form a linked list among themselves, are referred to as siblings of each other. Pointers are bi-directional: A parent points to the first of its children, and each child points back to its parent. Each sibling points to the sibling before ('before' means that the entity became a child earlier) it and the sibling after it. The last sibling points end around to the first. If there is only one child, its pointers point forward to itself and backwards to zero (a zero pointer). If there is more than one child, the first child points backwards to the last child, but with a negative pointer. All pointers are actually disk addresses, which give a page and a location on the page for an entity. (See Part III for details). With this type of structure it is entirely possible for an entity to have more than one parent. In this case we get a network or bi-directional tree. An entity can have several linked lists or rings passing through it, with each different linked list defined by a different parent. A parameter tells the system the maximum number of parents an entity can possess.

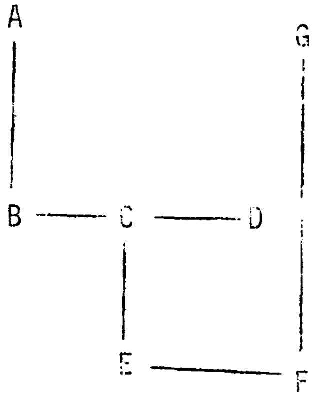
The file consists of two parts: a hash table and the data - entities and pointers. The hash table, the first part of the file, is like an index to the data. The name of an entity is hashed to a location in the table where the actual disk address of the entity is stored. The hash table is set up initially to a fixed length, while entities occur in the order they are added.

In general, it takes one disk access to get the hash entry for an entity, and another access to get the entity itself. But it is possible to store data in the hash entry so the only one access is necessary. Thirty bits are set aside for user flags in each hash entry. The thirty bits can either be set individually, or considered as an integer. There are also user flags with the pointers, which are described below.

User's Buffers - IBAF and IPNTR

Data - i.e., entity data and entity points - are transferred between the disk file and two user-supplied arrays: IBAF, the data buffer, and IPNTR, the pointer buffer. Their formats are shown in figure 3. Data put into IBAF is entered into the file with subroutine ADHASH and gotten from the file with the subroutine GETHSH. Word 1 gives the total amount of data in the file. Word 2 is the name of the entity, used by subroutine GETLOC to find a suitable hash location. Word 3 is the password, used to restrict GETHSHes. Words 4-n (n the value of IBAF(1)) are the data.

The IPNTR buffer is zeroed by ADHASH, except for word 1 which is returned as the entity location, and filled by GETHSH. Word 2 has flags to indicate whether or not this entity is a parent or child, and the number of parents the entity has, that is, the number of the three-word



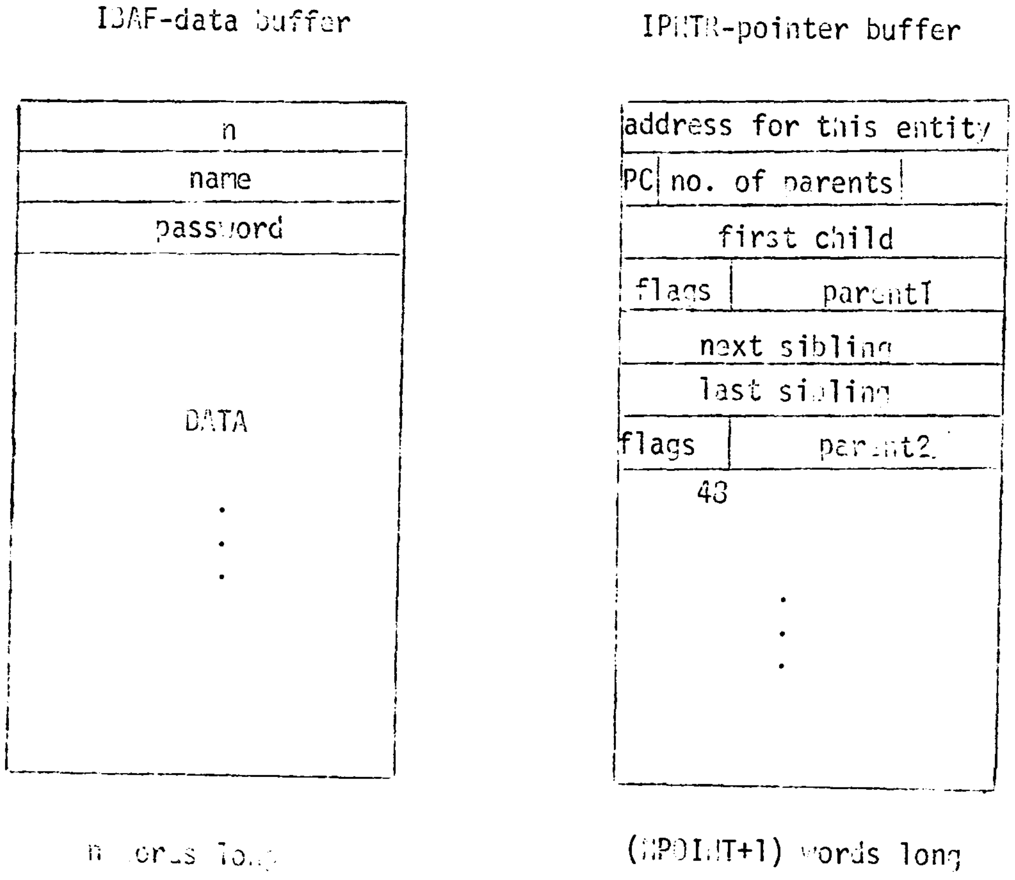
A is a parent of B, C, D

B, C, D are siblings, and children of A

E and F are children of C

F has two parents: C and G

Figure 2: Example of a Tree Structure



C-this entity a child

P-this entity a parent

Figure 3: Structure of IBAF and IPNTR

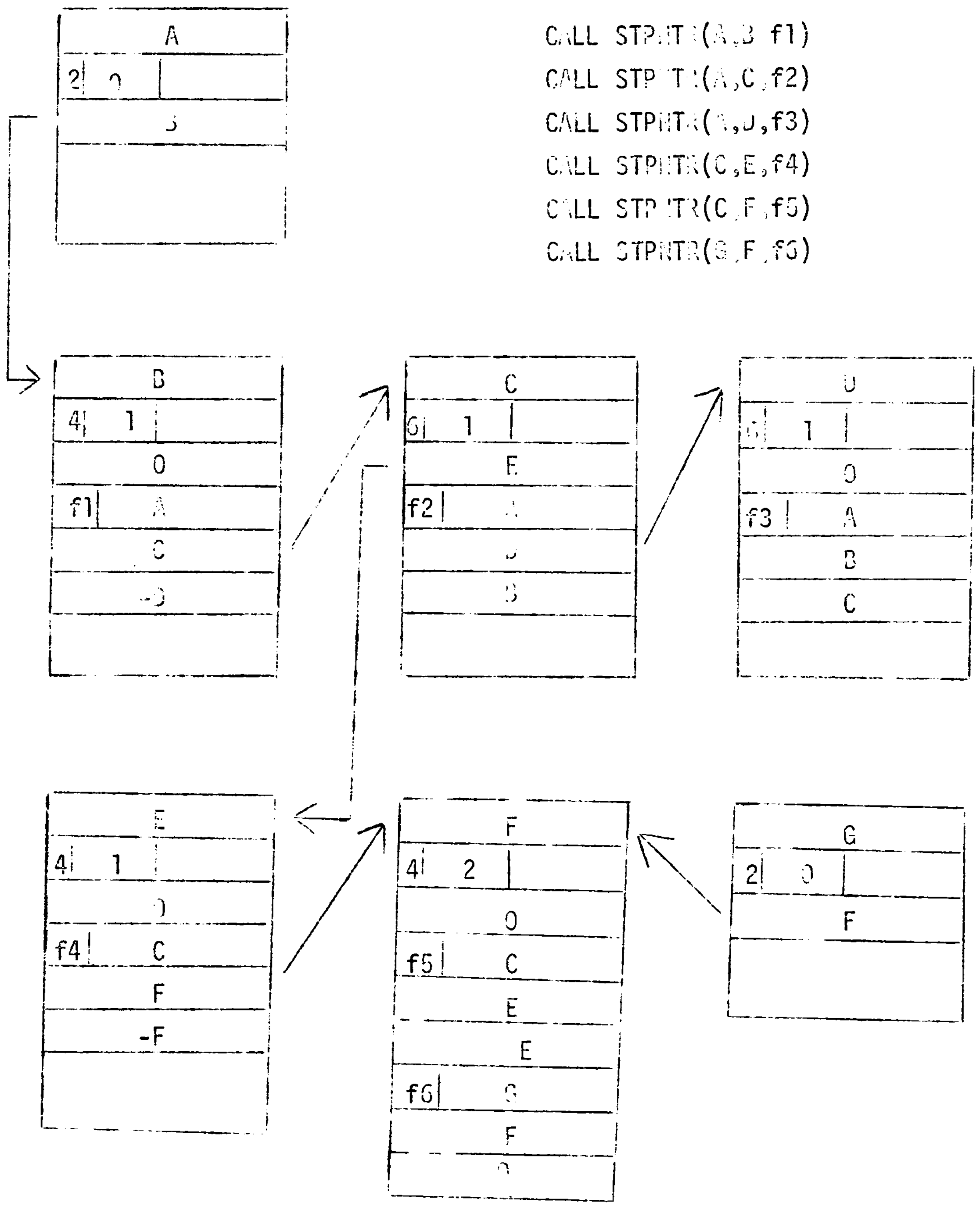
blocks which start in word 4. Word 3 points to the first child of this entity if it is a parent; else word 3 is zero. Following this is a three-word block for each parent the entity has. If no parents are allowed, IRNTR is just one word. The first word of the three points to the parent, the second word points forward to the next sibling on the linked list of children of the parent whose address is given in the first word, and the third word is a pointer to the previous sibling. The backwards pointer is zero if this is an only child, and is a negative address - complement to get the actual address of the last child - if this is the first child of a linked list.

Bits 48-58 of the first word of each block are 'pointer flags'. These can be used to differentiate children of the same parent - going horizontally along a linked list - or to differentiate parents of the same child - going vertically in the child. Used when moving up the tree from child to parent, the latter method will allow a user to determine which parent he wants without accessing all parents and checking some datum in them.

Using Overlays

In order to save core, three programs are used as overlays: a program to initialize the file, i.e., write numbered pages on the disk, a program to save data and pointers on tape in a compacted format, and a program to restore the file from the same tape. The first program is also available as a subroutine, but is usually not as fast as the overlay.

The use of these overlays with the FMS is different than the use of normal 6600 overlays. The user's main program runs as a primary overlay - and thus secondary overlays are allowed. When one of the three programs is called in, the user's program is written out on the disk and the desired overlay is read in as another primary overlay. When the overlay is finished,



```
CALL STPNTR(A,B f1)
CALL STPNTR(A,C,f2)
CALL STPNTR(A,D,f3)
CALL STPNTR(C,E,f4)
CALL STPNTR(C,F,f5)
CALL STPNTR(G,F,f6)
```

Figure 4: Pointers for Figure 2

the user's program is read back in from the disk and the program continues right after the spot the overlay was called. Thus an overlay can be called from anyplace in the user's program.

To initially form the overlays on a file certain control cards are needed.

These are:

```
JOB
FUN(S)
P,A,FLMLIB,UCC003.
P,A,SDN,VRN.
                         optional
P,O,OVLFL,SDN.
MODIFY(P=ISR,N=0,F)     $  EDIT,INITIAL
FUN(I=COMPILE,S)
LOAD,LGO,FMSLIB.
NOGO.
7-8-9
main overlay
6-7-8-9
```

SDN is a user-supplied subdirectory onto which the overlay file OVLFL is written. If the user doesn't wish to use a permanent file, OVLFL can be copied to tape after the NOGO card, and copied back to the disk when the overlay is going to be used.

The user can have his own overlays on OVLFL for calling in the same manner as above. To construct the overlay he adds his program preceded by the standard Fortran overlay card - OVERLAY(OVLFL,n,0), n greater than three - after the main overlay and before the end-of-file card in the above control card grouping. To call the overlay from the user's program, use CALL OVLPRO(n). The primary overlay and its associated subroutines - note that some subroutines, e.g., CPC, are loaded with the main overlay and need not be counted - must take up less than 44000B of core.

A main overlay program must be included before any primary overlay. The overlays generated above need not be changed unless the main overlay is changed, since its length determines where the primary overlays start. The length of the main overlay depends on the number of files used and the length of blank common. The format for the main overlay is:

```
OVERLAY(lfn,0,0)
PROGRAM MAIN (file 1, ...,file n)
COMMON IDUM(m)
CALL OVLAY(lfn,n,m)
END
```

lfn is the name of the user's file for his overlays - MAIN and his program. All files - e.g., INPUT,OUTPUT,TAPE27 - to be used in the user's main program or any of his overlays must be declared in the main overlay instead of the program where the files are used. Blank common should be declared in the main overlay as well as in the program it is used in. With no blank common used anyplace the COMMON card may be left out and m=0. As long as m - the length of blank common - and n - the number of files used - remain the same, the primary overlays do not have to be reconstructed.

II. USING THE SYSTEM

Initialization

SUBROUTINE OPNFLMG(N,LFN)

This must be the first FMS routine called. It opens the file for reading only if N=0, or for both reading and writing if N=1. LFN is the seven-character left-adjusted, zero-filled (i.e., L field) display code name for the file, to be entered into the first word of the file's FET.

SUBROUTINE INIT(ISSIZE,NPOINT,ISECPG,IHPG)

This routine should be called next if a new file is to be built. INIT calls the overlay INITIAL which writes out blank numbered pages.

ISSIZE is the number of pages set aside for the hash table at the beginning of the file, and is roughly proportional to the number of entities in the file. If there are N words per page, each page of the hash table has room for (N-2)/2 entities, or a total of ISSIZE*(N-2)/2 for the whole table. Depending on the chances of collision and how full one wishes the table to be, the total number of entities which can be entered in the hash table will be smaller than this, possible one-half to two-thirds of it. See discussion of the routines GETLOC and IHASH for more about hashing. ISSIZE can range from one to some number smaller than the size of the file.

NPOINT is the maximum number of parents any entity has. It is used to set aside the proper amount of space in each entity for possible pointers. NPOINT must be greater than or equal to zero; the only limits on NPOINT come from the limits on core and disk space. The user's pointer buffer IPNTR must be dimensioned at (NPOINT+1)*3 unless NPOINT=0, in which case IPNTR is a single word. NPOINT should be no larger than necessary to conserve space in the file.

The third argument, ISECPG, tells the system the page size in PRU's. Since on the disk the PRU's are 64 CM words long, the length of a page is 64*ISECPG words. Pages of more than one PRU are desirable when most of the entities are longer than 62 words. A little more efficiency can be achieved by making sure that most entites can fit on a page. But note that an entity can be less than or greater than a page long. There is no relationship between the length of a page and the length of an entity. An entity of length N(=IBAF(1)) requires NPOINT*2+N+1 words of storage, while each page can contain 64*ISECPG-1 words. Increasing ISECPG also increases the number of hash entries which can fit on a page.

IHPG is the file size, including ISSIZE hash pages, for the program INITIAL to write out. If an attempt is made to add an entity to a page greater than IHPG, the error message DISK FULL is placed in the dayfile and the program stops. IHPG should not be much larger than necessary because PP time is spent writing out the blank pages.

SUBROUTINE INIT1(ISSIZE,NPOINT,ISECPG,IHPG,ICOMSZ)

This routine does the same as INIT, except that the overlay INITIAL is not used. Instead, the labeled common area is used as a page buffer for writing out blank pages. Since the labeled common will probably not be as big as the buffer in INITIAL - 17,920 words - INIT1 will not be as fast. On the other hand, a separate overlay need not be made. The extra parameter gives the length of labeled common.

SUBROUTINE FILSLT(NSLT,IPGSLT,ICOMSZ,IDSZ,IPSZ)

This routine declares the value of various parameters used by the FMS. It should be called at the beginning of any program, including overlays, and after the return to the program from an overlay using the FMS. FILSLT can be used to change the slot size or the number of slots in the middle of a program, if called after a call to EMTSLT.

NSLT is the number of in-core slots - n in figure 1. For reasonable efficiency, NSLT should be at least three-one hash table slot and two entity slots. Depending on the structure of the file, timings may be improved by making NSLT bigger. If entities to be accessed close together, such as a parent and all its descendents, are widely separated on the disk - i.e., added at different times instead of sequentially - it would make sense to have many (small) slots so that the parents and all the children could be in core at one time. If, on the other hand, the entities are close together on the disk - such as a parent and its linked list of children, all of which were added at one time - it would be better to have (a few) large slots that can contain all the entities together.

IPGSLT is the number of pages in each in-core slot - m in figure 1. As mentioned above, IPGSLT can be varied to achieve maximum efficiency with minimum space for the labeled common.

ICOMSZ is the size of the labeled common /FMS/. This value is checked against the size needed for all the slots, and an error message is returned if ICOMSZ is too small. The length of labeled common is ICOMSZ = 25 + 64*ISECPG + NSLT*(64*ISECPG*IPGSLT+3) + 3*NOSLT. NOSLT is the number of ECS slots, and is zero if ECS is not used.

IDSZ and IPSZ are the length of IBAF and IPNTR, respectively. These are checked in ADHASH and GETHSH, and against NPOINT to avoid overrunning arrays. If IBAF is too small in GETHSH or IPNTR is too small, the program stops. If IBAF is too small in ADHASH - i.e., IBAF(1) is greater than IDSZ - a warning message is produced but the entity is added. If IDSZ or IPSZ are changed in the middle of a program, CALL SETSZ(IDSZ,IPSZ) is used to record the changes.

SUBROUTINE FILECS(NOSLT,NPGSLT)

This routine should be called - after OPNFLMG,INIT,FILSLT - only if the user's program will have ECS buffers. NOSLT is the number of ECS slots, and NPGSLT is the length of each ECS slot in number of pages. These two numbers can be varied for greatest efficiency much like NSLT and IPGSLT. If reasonably possible, it is very nice to have all the hash table in one ECS slot. The user, on his job card, must allow for at least NOSLT*NPGSLT*ISECPG*64 words of ECS. The maximum is 277000B or 97,792 words (1528 PRU). Labeled common must be increased by 3*NOSLT words to allow for an in-core ECS buffer header table. It is important to note that NPGSLT = k*IPGSLT, where k is some integer. If this were not the case, it would be possible to have duplicate pages in different in-core slots. Also, an in-core slot is used to transfer data from the disk to ECS. k must be an integer so that there will be an integer number of transfers. The smaller k is, of course, the fewer disk operations needed to fill each ECS buffer. If k is not an integer, an error message is entered in the dayfile.

SUBROUTINE PASWRD(IARAY)

IARAY is a nine-word array of seven-character left-adjusted passwords. When a GETHSH is performed, the password of the entity - in IBAF(3) - placed there before in ADHASH - is checked to see if it matches one of the nine passwords given in PASWRD. If it doesn't match, the program returns without the entity. In this way access of some entities can be denied to some users. If passwords are not used - i.e., PASWRD is not called - IBAF(3) can be used to store data in bits 18-59. If used, this should be the last of this group of subroutines to be called.

Record Manipulation

SUBROUTINE GETLOC (IBAF,IFLAG,LOC)

This routine will tell the user whether or not the entity with name in IBAF(2) is already in the file, and if not, where it can be added. This routine must always be called before ADHASH. Note that GETLOC does not change the file in any way. IBAF is usually a data buffer, but the only requirement on it is that the name of the entity be in IBAF(2).

After GETLOC hashes a name to get a number between one and ISSIZE, it makes a sequential search of that page for either an empty spot or a name matching IBAF(2). The page size determines the number of hash entries per page and the time necessary to search a page looking for either an empty entry or the entity desired. If neither condition is met cn that page a collision occurs and another page is fetched. This process continues until either: 1) twenty-four pages are checked or 2) the entity is found (has been entered previously) or 3) space is found to enter the entity. For case 1, IFLAG is returned as -1 and an error message is entered in the dayfile. For case 2 IFLAG=0 and LOC is the disk address where the entity is stored. For case 3 IFLAG=+1 and LOC is the hash address, used as an input to ADHASH, where the entity can be added. The chances for twenty-four collisions for a given name depend on a number of factors: if many of the names are similar, they are liable to hash to the same location. The larger the page the more hash entries it can hold. The bigger ISSIZE the more spread-out will be the entries in the hash table. The hash algorithm determines where the first and the next twenty-three hashes will be - see IHASH description below.

Occasionally the hash address may be desired for an entity already entered, e.g., in order to set some hash table flags. In this case IFLAG should be set to two before GETLOC is called. Then if IFLAG is returned as zero LOC will be the entity's hash address instead of its disk address.

FUNCTION IHASH(IBAF,ISSIZE)

FUNCTION IHASH1(IBAF,IH)

IHASH is called only by GETLOC. The function returns a hash number from one to ISSIZE, inclusive. This program has two entry points: IHASH(IBAF,ISSIZE) is called to get the first hash number. After a collision IHASH1(IBAF,IH) is called to determine the next hash number, where IH is the previous hash number. A standard IHASH is included with the FMS, but a user may wish to customize IHASH for his own file. There are two advantages to this: since the user knows which fields of his name are significant, he can hash to avoid collisions altogether.

Secondly, the user can see to it that related entities are stored together in the hash table, so that when one is accessed all related entities' hash entries are accessed also. In files with very large ISSIZE's customizing IHASH can lead to savings of 30% in CP time and 50% in PP time.

Consider one example where geographical data was being put in a file with the levels being state, county, township, section, quarter-section and parcel. It can be seen that in two counties there will be many forty-acre parcels. (In this particular instance there were about 80,000.) To hold all this information a large file was used: ISSIZE = 8280 and IHPG = 22000. If the normal hashing algorithm had been used, two contiguous parcels might find themselves at opposite ends of the hash table. Since 8280 page can't be held even in all of ECS, the number of disk accesses with this method would be tremendous. It was decided to put a township and all its associated children on thirty-six consecutive pages of hash table, with each page containing a section, its four quarter-sections and the related sixteen parcels. The name of each entity was designed so that its relationships could easily be determined: county number in (octal) digits 20-18, township number in digits 17-12, section number in 11 and 10, quarter-section in 9 and parcel number in 8-6. By adjusting ISSIZE so that the hashing algorithm would give a hash number which moved through the hash table four times but each time hitting a spot offset from the spot hit on the previous cycles, almost all collisions were avoided and ISSIZE wasn't excessively large.

In the above example one notices that each page was only two-thirds full. Greater use of each page could be achievedby doubling the page size and putting three sections per hash page. This resulted in a file shorter by about 20%, and the number of disk accesses was reduced by almost that much. Unfortunately, since the hash pages were twice as long extra CP time had to be spent in GETLOC searching to the bottom of the page to find some entities. As a result CP time increased by 36%.

Since IHASH is called so often, greater efficiency can be achieved by coding it in COMPASS. The standard IHASH is in COMPASS, but the following equivalent listing is in Fortran in order to be followed more easily.

```
      FUNCTION IHASH(IALF,ISSIZE)
      DIMENSION IALF(2),INUM(6)
      DATA (INUM=1,2,3,5,6,11),(MASK=1777B)
      IH=0  $  IB=IALF(2)
      DØ  1  I=1,6
      IH=IH + MOD(AND(IB,MASK)*INUM(I), ISSIZE)
    1  IB=LRSHFT(IB,-10)
      IHASH = MOD(IH,ISSIZE) + 1
      RETURN
      ENTRY IHASH1
C     IHASH1 is called whenever there has been a collision.
C     ISSIZE is the IHASH value where the collision occurred
        IHASH1 = ISSIZE + 1
C     In case of collision, try next page.
C     After calling IHASH1, GETLOC performs IF(IHASH1.GT.ISSIZE)
C      IHASH1=IHASH1-ISSIZE
      RETURN
      END
```

SUBROUTINE ADHASH(IBAF,IPNTR,LOC)

Subroutine ADHASH is used to add data to the disk file. The data added is taken from IBAF, with IBAF(2) being the name of the entity to be added and IBAF(3) its password. The entity's data - if any - is in IBAF(4) through IBAF(N), where N is the value of IBAF(1). The user must set IBAF(1) before an ADHASH. If N is greater than the previously declared data buffer length, a warning message is entered in the dayfile, but the addition continues. IPNTR is zeroed by ADHASH, but the disk address where the entity is located is returned in IPNTR(1). LOC is the hash address where the entity can be stored, as returned by GETLOC.

SUBROUTINE GETHSH(IBAF,IPNTR,IDL)

GETHSH is the workhorse of the FMS. This is the routine used to access the data and pointers of an entity already on the disk. An entity with disk address IDL is returned with data in IBAF and pointers in IPNTR - see figure 3 for formats of IBAF and IPNTR.

If IBAF(1) is greater than the declared data buffer size, an error message is entered in the dayfile and the program halts with a STOP 16. This is to prevent sections of core from being overwritten. If the password of the entity, IBAF(3), does not match any of the seven character passwords given in the subroutine PASWRD, an error message is given and GETHSH returns without the entity. This checking does not occur if PASWRD hasn't been called.

The disk address IDL can be obtained in a number of ways: an in-core table of addresses, filled by IPNTR(1) as returned by ADHASH could be used. Given the name, GETLOC will return the entity's address if it is already in the file. And of course an address may be obtained from the pointer buffer of another entity. In general, those methods that don't go through the hash table will be faster than those which do because an extra disk access is needed to read up the hash table.

SUBROUTINE RPHASH(IBAF,IPNTR)

This routine is used for altering an entity's data after it has been added to the disk. When RPHASH is called, the entity whose address is in IPNTR(1) has its data and pointers replaced by IBAF and IPNTR, respectively; therefore, IPNTR had better contain the entity's pointers. The most common way of assuring this is with a call to GETHSH with the desired entity before the call to RPHASH. The GETHSH call need not use the same data buffer as RPHASH, if the data is going to be changed considerably.

The length of the replacing data, IBAF(1), can be less than, equal to, or greater than the length of the replaced data. However, greater efficiency is achieved if the lengths are the same. In this case the pointers are not replaced and IPNTR need be just one word. If the user knows ahead of time that he is going to add data to an entity and make it longer some time in the future, he would be wise to add the entity initially with its longer length, and on each succeeding RPHASH just add the extra data without changing IBAF(1).

SUBROUTINE RMVHSH(IDL,IPNTR)

RMVHSH completely deletes the entity with disk address IDL from the file. The pointers of other entities are reconstructed as if entity IDL never existed. The hash cell of the entity becomes available for other use, and the space occupied by the entity is made empty. IPNTR is a scratch buffer NPOINT words in length. The pointer buffer can be used for this purpose, but that is not necessary.

Pointer Manipulation

Thus far the routines we have discussed could refer to an unorganized data file, and such is the case if NPOINT=0. But it is possible to construct relationships between the various entities. If NPOINT=1 we have a tree structure - every child has just one parent and the tree branches out from (usually) one central node to the leaves, which are the final descendents. If NPOINT is greater than one we have a more complex relationship - a network structure - where each entity can have more than one parent and more than one child. In fact, there is nothing to prohibit A having child B and B in turn having child A. The user should be very careful, however, on tree searches of circular structures of this kind, not to get into an infinite loop where A points to B, B points to A, A points to B, etc. Use of pointer flags could be helpful in the case. It should be pointed out that the order of parents is not necessarily consistent from entity to entity - i.e., one entity's first parent maybe another entity's second parent, depending on the order the pointers were set.

SUBROUTINE STPNTR(IDLP,IDLC,ITYPE)

This routine makes the entity with disk address IDLC a child of the entity with address IDLP, and adds pointer flag ITYPE to address IDLP in the child IDLC. (See figure 4.) IDLC is added as the last child of IDLP, and all necessary pointers are updated. The pointer flag can range from zero to 2**11-1 and obviously can have any significance the user desires. It should be pointed out that when doing address comparisons using a parent's address gotten from the first word of a three word pointer block, the pointer flags, if non-zero, should be removed before the comparison is done. STPNTR also sets flags in the hash table and word 2 of the pointers indicating that the entity is a parent and/or child.

SUBROUTINE SRTPTR(IDLP,IDLC,IDLN,ITYPE)

Occasionally, it is important to have the children of an entity sorted according to some criterion. For example, if the children represent water samples taken every month at the location represented by the parent, it would probably be useful to have the children arranged from oldest sample (first child) to latest sample (last child). Since STPNTR always adds the new child last on the linked list, if we came up with an out-of-order sample, it would be very time consuming to put the sample entity in the right place. In a case like this the routine SRTPTR would be used to insert the desired entity IDLC <u>before</u> IDLN (the next entity) as a child of IDLP. Note that IDLN is already a child of IDLP. The pointer flag in IDLC is again ITYPE.

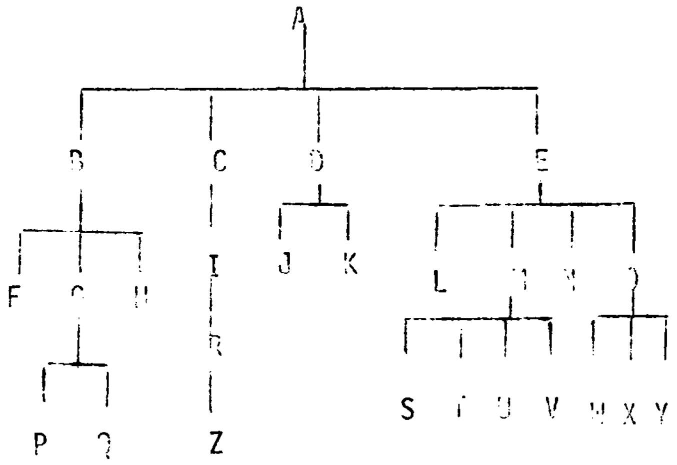


Figure 5a Tree Structure (NPOINT=1)

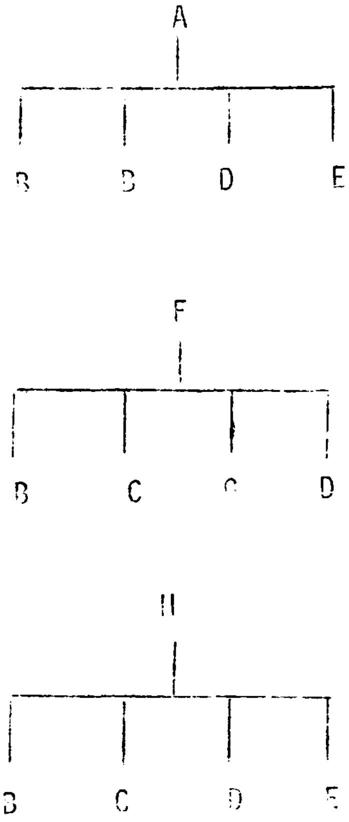


Figure 5b: Bi-directional Tree
(NPOINT=3)

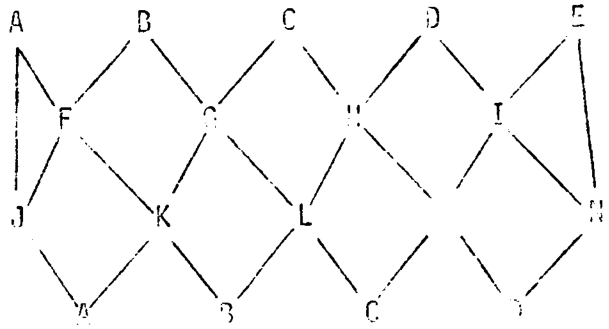


Figure 5c: Network (NPOINT=2)

A brief sample of coding should make the foregoing discussion more clear, and serve to illustrate some techniques for searching linked lists. Suppose our entities have their sort criterion in IBAF(5). The entities can have two parents, but as children of IDLP - its disk address - they should have IBAF(5) in monotonically decreasing order. Here is what we do when we wish to add another IDLC as a child of IDLP.

```
C         save IBAF(5) for IDLC
          ISAV=IBAF(5)
C         IDLC has already been added to the file
C         get IDLP
          CALL GETHSH(IBAF,IPNTR,IDLP)
C         get first child of IDLP (address in IPNTR(3))
          IF(IPNTR(3).EQ.0) GØ TØ 1
C         we know there is a first child; get it
          CALL GETHSH(IBAF,IPNTR,IPNTR(3))
C         search for block referring to IDLP
          DØ  2  I=4,7,3
          IF((IPNTR(I) .A.7777 7777B).EQ.IDLP) GØ TØ 3
        2  CØNTINUE
C         error
           STOP 10
C         see if this is an only child by checking backwards pointer
          3  IBK,IPNTR(I+2)
            IF(IBK.NE.0) GØ TØ 4
C         it is an only child.  see if IDLC should be before this entity
            IF(IBAF(5).GE.ISAV) GØ TØ 1
C           IDLC should be placed before this entity.  Save its address
            IDLN=IPNTR(1)
            GØ TØ 5
C         get last child.  IBK is now its address
        4  CALL GETHSH(IBAF,IPNTR,IBK)
C         check criterion
            IF(IBAF(5).GE.ISAV) GØ TØ 1
C         look at previous entity.  IDLN points to this entity.
        8   IDLN=IPNTR(1)
C         get proper parent
          DØ  6  I=4,7,3
          IF((IPNTR(I).A.7777 7777B).EQ.IDLP) GØ TØ 7
        6   CONTINUE
C         error
          STOP 11
        7   IBK=IPNTR(I+2)
C         check if this is the first child
            IF(IBK.LT.0) GØ TØ 5
C         get previous entity
          CALL GETHSH(IBAF,IPNTR,IBK)
C         check sort criterion
          IF(IBAF(5).LT.ISAV) GØ TØ 8
C         have found position - add child
        5   CALL SRTPTR(IDLP,IDLC,IDLN,0)
            GØ TØ 9
C         add IDLC as last child
        1   CALL STPNTR(IDLP,IDLC,0)
        9   CONTINUE
```

SUBROUTINE RMVPTR(IDLP,IDLC)

This routine does the opposite of the pointer - setting routines. It removes IDLC as a child of IDLP, and updates all necessary pointers. Flags in the hash table or word 2 of the pointers are changed if neccesary.

SUBROUTINE GTPNTR(IDL,ITYPE)

As seen in the previous example, the user can search the pointer buffer for addresses, since he knows exactly what he wants. No general pointer searching routine has been written, and the only routine for doing limited pointer searches is GTPNTR.

IDL is a three-word array. IDL(1) is always a parent's address, and should be filled by the user. If IDL(2) is zero then, on return from GTPNTR,IDL(2) is the address of the first child of IDL(1) - zero if no children - while IDL(3) and ITYPE are unchanged. If IDL(2) is non-zero when inputted, it is assumed to be the address of a child of IDL(1). GTPNTR then returns the forward pointer of IDL(2) - input value - in IDL(2) and the backwards pointer in IDL(3). ITYPE is the pointer flag for IDL(2) - input value - as a child of IDL(1). Note that GTPNTR destroys the original value of IDL(2). This routine does not require that the entity be fetched by a GETHSH, and so if only pointer checking is required, GTPNTR is faster than GETHSH and, in any case, it is never slower.

Miscellaneous Routines

1. Hash Table Flag Routines

Each hash entry contains thirty bits which are available to the user. These bits can either be set individually as flag 0 - flag 29, or as a thirty-bit integer. It is thus possible to store information in the hash table so that it is not necessary to access the data itself in all cases.

SUBROUTINE SETFLG(LOC,IBIT,M)

This routine sets bit IBIT(IBIT GE 0 and LE 29) to value M - either 0 or 1 - in entry with hash address LOC, as returned from GETLOC with IFLAG initially two.

SUBROUTINE GETFLG(LOC,IBIT,M)

This routine is the complementary one to SETFLG. It returns as M the value of bit IBIT in hash entry LOC. If IBIT=33,M will be one if the entity is a child; if IBIT=34, M will be one if the entity is a parent. These two bits are set by the pointer routines.

SUBROUTINE STINFG(LOC,IVAL)

This routine sets the integer IVAL - IVAL GE 0 and LT 2**30 - into hash entry LOC.

SUBROUTINE GTINFG(LOC,IVAL)

This routine returns the integer IVAL from hash entry LOC as was set by STINFG or, bit by bit, by SETFLG.

2. Debugging Aids

SUBROUTINE IPRINT(N,IBUF)

SUBROUTINE GO

SUBROUTINE STP

Subroutine IPRINT causes N numbers - IBUF(1) through IBUF(N) - to be entered in the dayfile. The dayfile has a message limit of about 200, after which the program aborts. Each number is in octal. Initially IPRINT is enabled. To disable IPRINT, use CALL STP. To re-enable it, use CALL GO. All calls to IPRINT between CALL STP and CALL GO will cause nothing to happen.

FUNCTION NERR(I)

NERR(I) - where I is a dummy variable - returns the value of the last FMS error (see list of error messages for corresponding numbers) andsets the error number to zero. If NERR(I) is zero, no errors have occured since the last call to NERR or since the beginning of the program.

SUBROUTINE SCOOP(IFG)

Subroutine SCOOP prints out the tree structure of the file. It does this by searching through the hash table for all unattached entities and printing them out. Then it searches for the first entity that is a parent but not a child; it prints out this name and then searches for the entity's children, and so on until the whole tree has been printed out. The next entity which has children but no parents is searched for, and the process continues. See figure 13 for a sample of SCOOP output. If IFG=0 the name - IBAF(2) - is considered to be made up of ten display codes, and is printed in an A10 field. If IFG=1 the name is printed out in an Ø20 field. When SCOOP is used, an OUTPUT file must be declared on the program card. This is the only FMS routine which requires a declared file.

End-Of-Job Processing

SUBROUTINE EMTSLT

Subroutine EMTSLT makes sure that all altered data in core or in ECS are written onto the disk file, so the file is brought up to date. It can be called anytime. EMTSLT must be called at the end of a main program if user wishes to save his data; it should be called before exiting to an FMS overlay (except for the three system overlays) and just before returning from an overlay.

SUBROUTINE CLOSE

This routine calls EMTSLT and then sets the file to closed status. The routine needn't be called unless the file is permanent.

SUBROUTINE SAVE(LFN)
SUBROUTINE RESTORE(LFN)

These are complementary subroutines, each calling an overlay. SAVE is called at the end of a program to save the data on a tape with left-adjusted zero-filled name LFN. RESTORE is called at the beginning of a program, after INIT, to restore a file from a tape made by SAVE. Unlike the programs discussed further on SAVE/RESTORE do not merely copy the file onto tape, but act as a garbage collector. RESTORE adds entities and sets pointers; any deleted entities will be forgotten, and entities will be sorted according to linked list membership. In other words, after a SAVE/RESTORE all children of a given entity will be consecutive in the file; this is done to increase access speed. SAVE asks for card input. The first card read in has an I5 field telling how many more cards there are. If this integer is zero, SAVE starts with the first entity in the hash table with children but no parents, and stores it and all its children, then all the children of the first child, then all the children of the first grandchild, etc. No entity is stored twice. If an entity is the child of more than one parent, it is stored with the children of the first parent which occurs in the search of the hash table. This may not always be the most useful order. If the integer read in is not zero, the next cards each contain the name of an entity in an Ø20 field. That entity is stored, followed by its linked list of children, and so forth. After all cards are processed, the remainder of the entities are stored as described above. If other cards are used as input to the main program, the cards for SAVE should be separated from the others by a 7-8-9 card. Since SAVE and RESTORE use the FMS, they are not as fast as the routines described below. Both overlays need a EC277 on the job card.

SAVE destroys passwords.

SUBROUTINE QDSAV(ICOMSZ,LFN)

SUBROUTINE QDRST(ICOMSZ,LFN)

These subroutines serve to save and restore a file on tape with name LFN, but unlike the previous routines, no file compaction or alteration of order is done. The file is copied to/from tape, using labeled common - length ICOMSZ - as the transfer buffer. These routines are faster than the previous ones, and don't use overlays, but require much more tape.

III. INTERNAL DESCRIPTION OF THE SYSTEM

The following section describes in detail some of the internal workings of the system. Knowledge of the topics discussed is not necessary in order to use the system.

Constant Storage In Labeled Common

Figure 1 was a diagram of the layout of labeled common. The first twenty-five words of /FMS/ hold constants used in the routines which make up the FMS. A brief description of each constant follows (refer to figure 6):

ISSIZE is the size of the hash table, in pages. It is input through INIT or INIT1.

NPOINT is the maximum number of parents an entity may have. It is input through INIT or INIT1.

NBAF is the length of the data buffer, IBAF. It is input through FILSLT or SETSZ.

IWDPG is the number of words per file page. It is equal to 64*ISECPG.

LSTPG is the last page data has been entered on. Before any data is entered, but space for the hash table has been set aside, LSTPG= ISSIZE+1. LSTPG is updated through ADHASH in routine PGFIND - the routine that inds a page with available space.

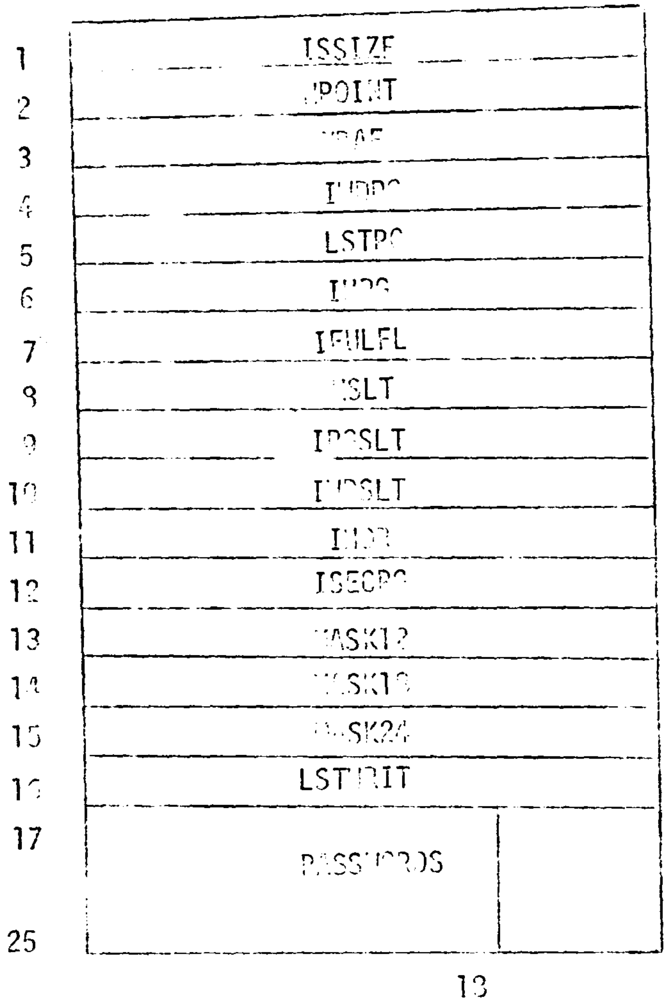
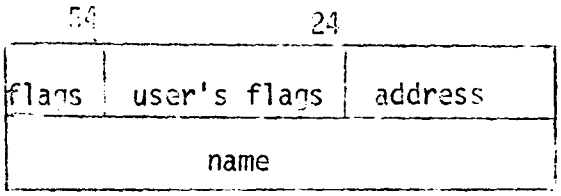


Figure 6: First 25 Words of Labeled Common



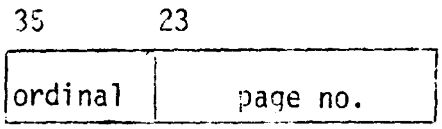


Figure 7a: Hash Entry

Figure 7b: Hash Address
(as returned by GETLOC)

IHPG is the highest page of the file. It is input through INIT or INIT1.

IFULFL is a flag used by the system to indicate that any available space can be used for data storage, and not just a minimum of six words, which is the case if IFULFL=1. It also indicates the number of passes through the file looking for free space. PGFIND, after going through the file and not finding any available space, will go back through the file and look for space not filled before, or from which entities have been deleted. IFULFL is one on the first pass and is doubled for each succeeding pass until IWDPG/IFULFL.LT.6.

NSLT is the number of in-core slots. It is input through FILSLT.

IPGSLT is the number of pages in each in-core slot. It is input through FILSLT.

IWDSLT is the length of a slot. It is equal to IWDPG*IPGSLT+IHDR.

IHDR is the number of header words per slot. In the current system this value is three.

ISECPG is the number of 64-word PRU's per page. It is inputted through INIT or INIT1.

MASK12,MASK18,MASK24 are often used masks. Their values are 7777B, 777777B and 77777 7777B, respectively.

LSTWRIT is the last PRU on which anything has been written. Its value should be, if a SAVE has not been done, IHPG*ISECPG+2. The '2' occurs because the first PRU is used to hold a copy of the first twenty-five words of labeled common, and the last PRU is an end-of-record.

The nine passwords occupy bits 18-59 of the next nine words. The bottom eighteen bits can be used for passing values between routines.

Arrangement of the Hash Table

When the file is initialized, ISSIZE pages are set aside at the beginning of the file for the hash table. When an entity is hashed to a page, it is entered on that page in a two-word entry as near the top as possible. In other words, entries start at the top and fill it downward. When the page is full, any further hashes to it are considered collisions. The structure of the two word hash entry is illustrated in figure 7a. The flags' significance are:

bit 59 set - this cell in use
bit 58 set - this entity a parent
bit 57 set - this entity a child
bit 56 set - entity deleted
bit 55 set - this entity saved in SAVE
bit 54 set - this entity's children saved in SAVE

Bit 56 is used because GETLOC, when looking for an entity already entered, stops at the first empty cell; a deleted hash entry must not look the same as an empty cell. If the entity in question is not found, it will be put in a deleted entry location if such exists.

Data Storage On A Page

The appearance of a file page is shown in figure 8. Each page hold up to sixty-three records, i.e., parts of entities. The first word on each page contains the page number - put there by INIT - the current number of records on that page and the space remaining on that page for adding further records. Following this header word are the record directories, one per record. Each directory word contains the length of the record, its location on the page relative to the start of the directories, and a pointer to the next portion of the entity - zero if this record is the end of the entity - on another page. The records themselves are stored starting at the bottom.

When an entity is accessed using GETHSH, both the data and pointer buffers are filled. On the disk the data from these two buffers is stored together. As can be seen from figure 9, the password and lengths are packed together and the pointers are stored between the name and the rest of the data. The pointers are packed so that for a pointer buffer of length (NPOINT+1)*3 only NPOINT*2+1 words of disk storage are used.

When data is transferred between an in-core slot and a user's buffer, the intermediary is the page buffer (see figures 1,10). The first word of the page buffer is its length. This is one more than the record's length, as stored in the record directory. Then comes the data from each disk page. At the end is the continuation pointer to the next page of data; this pointer is taken from the directory also.

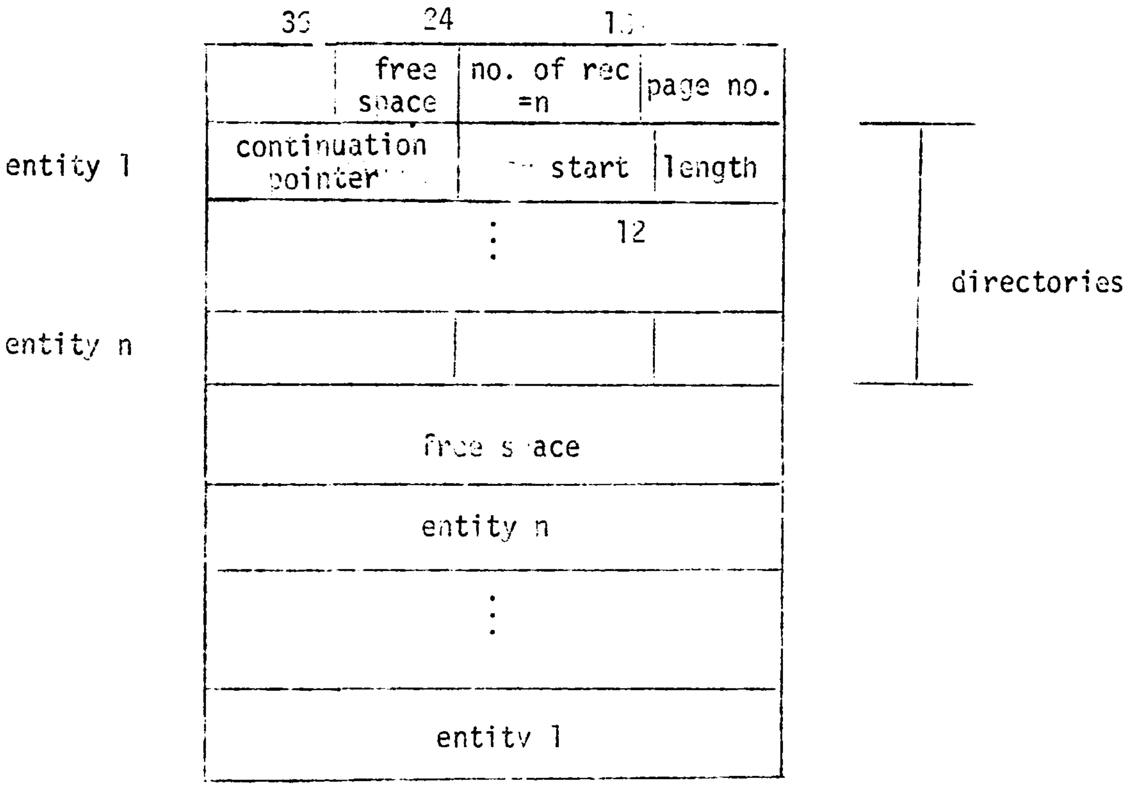


Figure 8: Layout of a Page

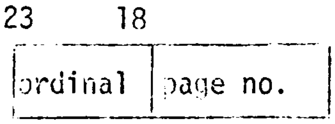


Figure 11: Disk Address (Format of all Pointers)

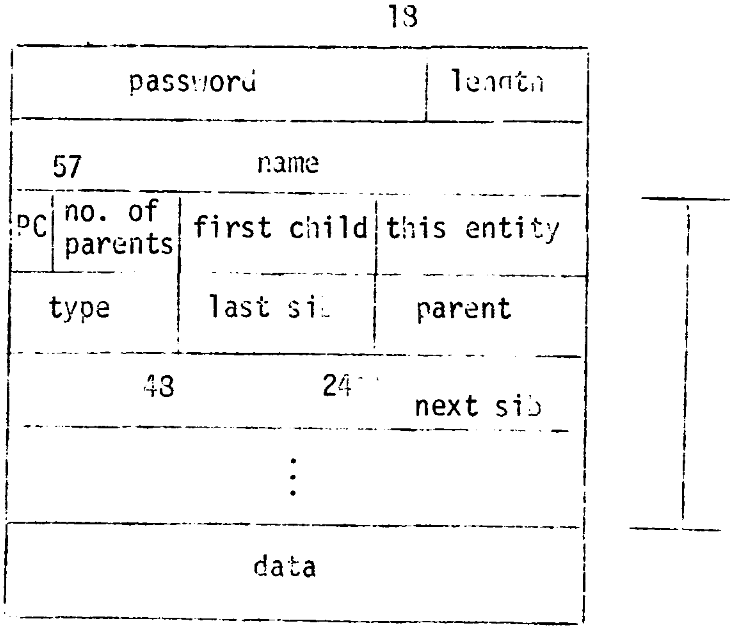


Figure 9: Data Storage on Disk

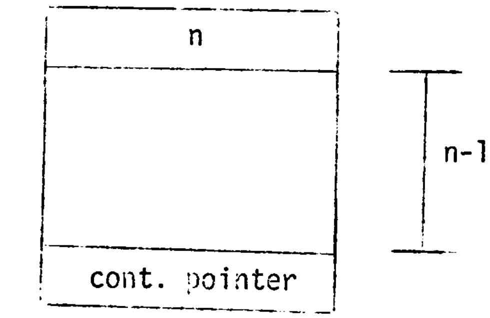


Figure 10: Page Buffer

Algorithm For Filling Slots (Subroutine PAGE)

The labeled common area of the user's program consists of - for the most part - a series of slots which hold page images from the disk. Each slot has a three-word header made up of 1) page number of first page in that slot, 2) age of the slot, and 3) number of times the slot has been changed. A slot is indicated as changed whenever something is entered or replaced on a page in that slot. The following description shows how the page is found if in core, and if it isn't which slot the page is read into. The exact same mechanism is used for the ECS slots:

1) Is the page desired in the last slot handled on the previous call to PAGE?

If not:

2) If word 1 of the header is zero, read the page into this slot.

If not:

3) Is page in this slot?

If not:

4) If this slot is unchanged - header word 3=0- and there were no unchanged slots previously, save location and age, and go to 8.
5) If this slot is the oldest unchanged slot - when there is more than one - save location and age of this slot, and go to 8.
6) If this slot has been changed, but an unchanged slot has been found previously, go to 8.
7) If all of the slots including this one have been changed, but this is the oldest slot, save location and age of this slot.
8) Increase age of this slot by one. If there are any more slots, go to 2.

At the end of this process either we have the page desired or else we have the oldest unchanged slot, if it exists, and we can read the page desired into it. Otherwise we write out the oldest changed slot and read the page desired into it.

IV. SAMPLE PROGRAM

The following sample programs set up a data base for the two-story building shown in figure 12. Data is then read in from cards and added to the entities. Finally, a search is made through the file to determine the thickness of the thinest interior wall which surrounds each room. This information is stored in word 4 of each room entity. The structure of the file is shown in figure 13, as produced by a call to SCOOP.

The first, short, program sets up the overlays for the main program to use. The second program consists of the main overlay and the primary overlay, PROGRAM EXAMPLE, which constructs and manipulates the file. The second program took 1.326 seconds to compile and load, and ran for .516 seconds.

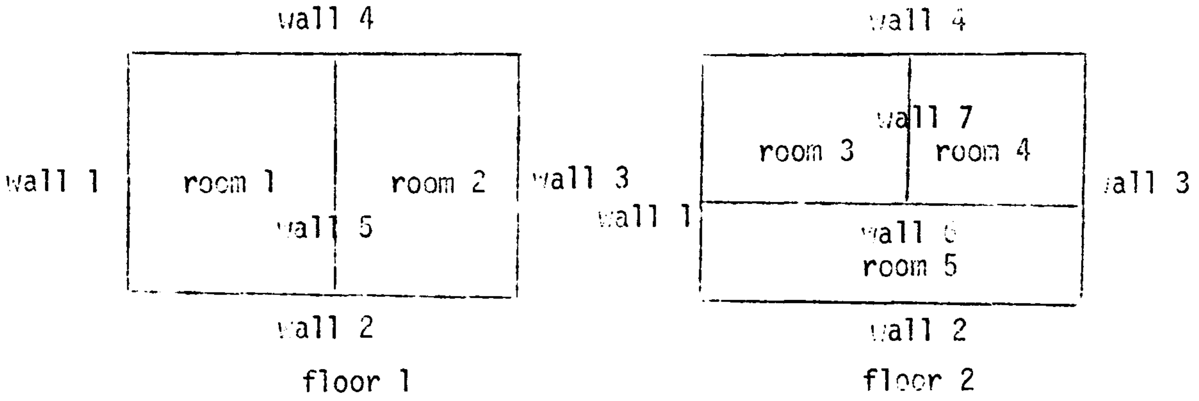


Figure 12: File Set Up By Sample Program

```
BUILDING1
        FLOOR1
                ROOM1
                        WALL1
                        WALL2
                        WALL4
                        WALL5
                ROOM2
                        WALL2
                        WALL3
                        WALL4
                        WALL5
        FLOOR2
                ROOM3
                        WALL1
                        WALL4
                        WALL6
                        WALL7
                ROOM4
                        WALL3
                        WALL4
                        WALL6
                        WALL7
                ROOM5
                        WALL2
                        WALL3
                        WALL6
                        WALL1
```

Figure 13: Structure of File Set Up By Sample Program

```
JOB,CM60000,T5.12345678
COMMENT.THESE CONTROL CARDS SET UP FMS OVERLAYS ON FILE OVLFL.
P,A,FLMLIB,UCC003.
FUN(S)
D,OVLFL.
P,D,OVLFL,FLMLIB.
MODIFY(P=ISR,N=0,F) $ EDIT,INITIAL
FUN(S,I=COMPILE)
LOAD,LGO,FMSLIB.
NOGO.
7-8-9
      OVERLAY(NAME,0,0)
      PROGRAM MAIN(INPUT,OUTPUT)
      COMMON IDUM(80)
      CALL OVLAY(4HNAME,2,80)
      END
6-7-8-9
```

```
JOB,T5,CM75000,EC32.12345678
COMMENT. NOTE THAT WHILE THE PRIMARY OVERLAY IS ONLY 20K LONG,
COMMENT. ALLOWANCES MUST BE MADE FOR THE 44K PRIMARY OVERLAY INITIAL.
P,A,FLMLIB,UCC003.
FUN(S)
LOAD,LGO,FMSLIB.
COMMENT.REPLACE NEXT TWO CARDS WITH 'EXECUTE.' CARD IF NOT USING OVERLAYS
NOGO.
NAME.
7-8-9
      OVERLAY(NAME,0,0)
      PROGRAM MAIN(INPUT,OUTPUT)
      COMMON IDUM(80)
      CALL OVLAY(4HNAME,2,80)
      END
      OVERLAY(NAME,1,0)
      PROGRAM EXAMPLE
C     3310 = 25+64+5*(10*64+3)+2*3
      COMMON/FMS/IDUM(3310)
      COMMON IBAF(50),IPNTR(21),IARAY(9)
      DIMENSION IFL(2),IRM(5),IWL(7),IDAT(25)
      DO 1201 I=1,9
 1201 IARAY(I)=0
C     OPEN FILE(NAME FILEMNG) FOR BOTH READING AND WRITING
      CALL OPNFLMG(1,7LFILEMNG)
C     USE A HASH TABLE 50 PAGES LONG IN A TOTAL FILE OF 500 PAGES
C     ALLOW 6 PARENTS PER RECORD
C     EACH PAGE IS ONE PRU
      CALL INIT(50,6,1,500)
C     5 SLOTS, 10 PAGES PER SLOT
      CALL FILSLT(5,10,3310,25,21)
C     2 ECS SLOTS, 50 PAGES PER SLOT.  NOTE THAT 10 DIVIDES 50 AND
C     ALL OF THE HASH TABLE CAN BE KEPT IN ONE ECS SLOT
      CALL FILECS(2,50)
      ITYP=0
C     STORE PASSWORDS
C     THIS CALL NOT NEEDED IF PASSWORDS NOT USED
      CALL PASWRD(IARAY)
C     SET UP TREE STRUCTURE, EACH RECORD 25 WORDS LONG
      IBAF(1)=25
C     PASSWORD=0
```

```
      IBAF(3)=0
C     NAME FOR BUILDING RECORD
      IBAF(2)=10HBUILDING1
      DO 1 I=4,25
    1 IBAF(I)=0
C     FIND EMPTY LOCATION IN HASH TABLE
      CALL GETLOC(IBAF,IPNTR,LOC)
C     ADD RECORD
      CALL ADHASH(IBAF,IPNTR,LOC)
C     SAVE ADDRESS OF BUILDING
C     ADDRESS WHERE ENTITY STORED RETURNED BY ADHASH IN IPNTR(1)
      IBLD=IPNTR(1)
C     ADD TWO FLOOR RECORDS, MAKE THEM CHILDREN OF THE BUILDING RECORD
      IBAF(2)=10HFLOOR0
      DO 2 I=1,2
C     FORM CORRECT NAME
      IBAF(2)=IBAF(2)+64**4
      CALL GETLOC(IBAF,IFLAG,LOC)
      CALL ADHASH(IBAF,IPNTR,LOC)
      CALL STPNTR(IBLD,IPNTR(1),ITYP)
C     STORE FLOOR ADDRESSES FOR FUTURE REFERENCE
      IFL(I)=IPNTR(1)
    2 CONTINUE
      IBAF(2)=10HROOM0
      CREATE,ADD,LINK ROOMS
      DO 3 I=1,5
      IBAF(2)=IBAF(2)+64**5
      CALL GETLOC(IBAF,IFLAG,LOC)
      CALL ADHASH(IBAF,IPNTR,LOC)
C     GET CORRECT PARENTS ADDRESS
      IP=IFL(2)
      IF(I.LE.2) IP=IFL(1)
      CALL STPNTR(IP,IPNTR(1),ITYP)
      IRM(I)=IPNTR(1)
    3 CONTINUE
      IBAF(2)=10HWALL0
C     FINALLY, ADD THE WALLS
      DO 4 I=1,7
      IBAF(2)=IBAF(2)+64**5
      CALL GETLOC(IBAF,IFLAG,LOC)
      CALL ADHASH(IBAF,IPNTR,LOC)
    4 IWL(I)=IPNTR(1)
C     SET PROPER POINTERS TO WALLS
C     POINTER TYPE IN WALL INDICATES ROOM NUMBER
      CALL STPNTR(IRM(1),IWL(1),1)
      CALL STPNTR(IRM(1),IWL(2),1)
      CALL STPNTR(IRM(1),IWL(4),1)
      CALL STPNTR(IRM(1),IWL(5),1)
      CALL STPNTR(IRM(2),IWL(2),2)
      CALL STPNTR(IRM(2),IWL(3),2)
      CALL STPNTR(IRM(2),IWL(4),2)
      CALL STPNTR(IRM(2),IWL(5),2)
      CALL STPNTR(IRM(3),IWL(1),3)
      CALL STPNTR(IRM(3),IWL(4),3)
      CALL STPNTR(IRM(3),IWL(6),3)
      CALL STPNTR(IRM(3),IWL(7),3)
      CALL STPNTR(IRM(4),IWL(3),4)
      CALL STPNTR(IRM(4),IWL(4),4)
      CALL STPNTR(IRM(4),IWL(6),4)
      CALL STPNTR(IRM(4),IWL(7),4)
      CALL STPNTR(IRM(5),IWL(2),5)
      CALL STPNTR(IRM(5),IWL(3),5)
      CALL STPNTR(IRM(5),IWL(6),5)
      CALL STPNTR(IRM(5),IWL(1),5)
```

```
C     PRINT OUT TREE STRUCTURE
      CALL SCOOP(0)
C     NEXT, READ IN DATA CARDS, WHICH HAVE NAME OF THE RECORD IN COLUMN
C     1-10, DATA IN THE REST OF THE COLUMNS, AND ADD DATA TO ALREADY
C     EXISTING RECORDS
  100 FORMAT (A10,20I3)
      DO 5 I=1,15
      READ 100,IBAF(2), (IDAT(J),J=4,23)
C     NAME IN IBAF(2).  GET RECORD AND ADD DATA
      CALL GETLOC(IBAF,IFLAG,IAD)
C     SINCE ENTITY ALREADY PRESENT (IFLAG COULD BE CHECKED IF WE WEREN'T
C     SURE), IAD IS RETURNED AS THE ADDRESS OF THE ENTITY
C     GET THE RECORD INTO IBAF AND IPNTR
      CALL GETHSH(IBAF,IPNTR,IAD)
C     ADD DATA TO RECORD, REPLACE IT IN FILE
      DO 6 J=1,3
    6 IDAT(J)=IBAF(J)
      IDAT(24)=IBAF(24)
      IDAT(25)=IBAF(25)
      CALL RPHASH(IDAT,IPNTR)
    5 CONTINUE
C     THE CODING BELOW ILLUSTRATES HOW SEARCHING OF LINKED LISTS CAN
C     BE DONE.  WE COULD ALSO USE THE ADDRESSES STORED IN ARRAYS IFL,IRM,IWL
C     GET BUILDING
      IDAT(1)=IBLD
      IDAT(2)=IDAT(3)=0
      CALL GTPNTR(IDAT,ITYP)
C     IDAT(2) IS RETURNED AS FIRST CHILD OF IBLG, I.E. FLOOR1
      IFL1=IDAT(2)
      SAVE ADDRESS OF FIRST FLOOR RECORD
   17 IDAT(1)=IDAT(2)
C     SAVE ADDRESS OF THIS FLOOR
      IFLN=IDAT(2)
      IDAT(2)=0
      CALL GTPNTR(IDAT,ITYP)
C     IDAT(2) IS ADDRESS OF FIRST ROOM ON THAT FLOOR
      IR=IRM1=IDAT(2)
   18 IDAT(1)=IR
      IDAT(2)=0
      CALL GTPNTR(IDAT,ITYP)
C     IDAT(2) IS THE ADDRESS OF THE FIRST WALL OF THE ROOM
C     INITIALIZE MINIMUM
      MINW=1000 000
C     GET FIRST WALL
      IWL1=IDAT(2)
   19 CALL GETHSH(IBAF,IPNTR,IDAT(2))
C     WORD 5 IS POSITIVE IF THE WALL IS EXTERIOR
C     WORD 5 IS NEGATIVE IF THE WALL IS INTERIOR
C     THICKNESS OF INTERIOR WALL IS IN WORD 6
      IF(IBAF(5).GT.0) GO TO 29
      MINW=MIN0(MINW,IBAF(6))
C     SEARCH POINTER BUFFER FOR NEXT CHILD
   29 DO 20 I=4,20,3
C     LOOK FOR PARENTS ADDRESS (IR) IN THE POINTER BUFFER OF THE CHILD
C     THERE ARE AT MOST SIX PARENTS OF EACH WALL
      IF((IPNTR(I).A.7777 7777B).EQ.IR) GO TO 25
   20 CONTINUE
      PRINT 1000, IDAT(1),IDAT(2)
 1000 FORMAT(* PARENT *,O20,* NOT FOUND IN *,O20)
      STOP 40
C     GET ADDRESS OF NEXT CHILD ON LINKED LIST
C     IPNTR(I+1) IS FORWARD POINTER
   25 IDAT(2)=IPNTR(I+1)
C     CHECK IF ITS THE FIRST CHILD
```

```
      IF(IDAT(2).NE.IWL1) GO TO 19
      FINISHED WITH ALL WALLS SURROUNDING THAT ROOM
C     GET ROOM, ADD THICKNESS OF THINEST INTERIOR WALL SURROUNDING IT AS IBAF(4)
      CALL GETHSH(IBAF,IPNTR,IR)
      IBAF(4)=MINW
      IF(MINW.EQ.1000 0000B) IBAF(4)=0
C     REPLACE ROOM RECORD IN FILE
      CALL RPHASH(IBAF,IPNTR)
C     ROOMS HAVE ONE PARENT ONLY
      IR=IPNTR(5)
C     GET NEXT ROOM.  IF ITS NOT THE FIRST, LOOP AND CHECK ITS WALLS
      IF(IR.NE.IRM1) GO TO 18
C     GET NEXT FLOOR
C     ADDRESS OF PARENT (BUILDING)
      IDAT(1)=IBLD
C     ADDRESS OF A CHILD (FLOOR)
      IDAT(2)=IFLN
C     GET POINTER TO NEXT CHILD
      CALL GTPNTR(IDAT,ITYP)
C     IF ADDRESS OF NEXT FLOOR IS THE SAME AS ADDRESS OF FIRST FLOOR,
      WE HAVE CHECKED ALL FLOORS
      IF(IDAT(2).NE.IFL1) GO TO 17
C     ALL FLOORS FINISHED
C     WRITE INFORMATION FROM IN-CORE (AND ECS) SLOTS BACK ONTO THE DISK
      CALL EMTSLT
      STOP 500
      END
7-8-9
DATA
6-7-8-9
```

GLOSSARY

DATA BUFFER(IBAF) - a buffer used to hold data for adding to the file, and to receive data when accessing the file. Length is in word 1, name in word 2, password in word 3. Minimum length is three; maximum length not greater than IDSZ, which is a user-supplied parameter. See figure 3.

DISK ADDRESS(IDL) - a two-part address (see figure 11) used as a pointer to an entity. It indicates the page the entity is on, and where on that page it is. This is one of the three ways of referring to an entity, the other two being name and hash address.

ENTITY - a fixed length data bead. The file - except for the hash table - consists of a large number of entities and their associated pointers.

HASH ADDRESS(LOC) - a two-part address (see figure 7b) used as a pointer to the hash entry of an entity. It is returned by GETLOC if an entity can be added to the file.

HASH TABLE - a portion of the file ISSIZE pages long acting as an index to the rest of the file, where the entities themselves are stored. Each page of the hash table is filled, top-down, with a two-word hash entry (see figure 7a) for each entity added to the file.

ICOMSZ - the length of the user's labeled common,/FMS/. ICOMSZ=25+ ISECPG*64+NSLT*(IPGSLT*ISECPG*64+3)+NOSLT*3.

IFLAG - a flag returned by GETLOC showing the status of the hash table: IFLAG=-1, hash table overflow, no more room, IFLAG=0, entity present with its disk address returned - unless IFLAG=2 on input, IFLAG=1 entity not present but can be entered at returned hash address.

IHPG - the largest page of the file. It must be greater than ISSIZE.

IPGSLT - the number of pages per in-core slot in /FMS/.

ISECPG - the number of 64-word disk PRU's per file page. It must be between one and twenty-eight, inclusive.

ISSIZE - the number of pages set aside at the beginning of the file for the hash table.

NOSLT - the number of ECS slots. This value is automatically zero if ECS is not used.

NPGSLT - the number of pages in each ECS slot. It must be evenly divisible by IPGSLT.

NPOINT - the maximum number of parents an entity can have. Its value must be greater than or equal to zero.

NSLT - the number of in-core slots. It should be at least two.

PAGE - the standard unit of the file. A page can range from 64 to 1792 CM words long.

PASSWORDS - nine seven-character words which can be used to restrict access to certain entities.

POINTER - the disk address where an entity is located.

POINTER BUFFER(IPNTR) - a user-supplied buffer of length (NPOINT+1)*3 (or 1 if NPOINT=0) that holds an entity's pointers - after a GETHSH - which show that entity's relationships to other entities. See figure 3.

POINTER FLAG(ITYPE) - a number ranging from 0 to 3777B (2047) which appears in bits 48-58 of the parent's pointer in the pointer buffer of a child of that parent.

SLOT - a grouping of one or more pages in the labeled common area. Since file to in-core I-Ø is always done on a slot basis, each slot contains consecutive pages.

ERROR MESSAGES

Error messages are output to the dayfile. After the error, return is to the calling program without performing the desired action unless otherwise stated. Messages are listed below in alphabetical order, followed by the NERR number, followed by probable cause and/or action.

BAD I/O - 13
Problems with disk I/O - hardware difficulties

BAD PARAMETERS IN FLAG - 17
In SETFLG or GETFLG either the bit count or bit value is wrong. If value is incorrect, only necessary bits - either 1 or 30 - are used.

CANT FIND PARENT - 23
Pointer has not been set.

CANT FIND RECORD - 26
In SAVE or RESTORE, a record cannot be found.

DATA BUFFER SIZE TOO SMALL - 16
An attempt was made to add or retrieve a record whose length - specified in word 1 of the data buffer - was larger than the length of the data buffer - specified in FILSLT or SETSZ. If error occurs in GETHSH, program terminates - STOP 16. If it occurs in ADHASH, adding continues.

DISK FULL - 6
No room in file to store additional data. Program terminates with STOP 6.

DONT HAVE CORRECT PASSWORD - 18
Attempt to retrieve a record whose password was none of the nine user's passwords.

FILE NOT OPENED - 14
OPNFLMG not called

HASH TABLE OVERFLOW - 24
IFLAG=-1 in GETLOC

ILLEGAL ECS BUFFER LENGTH - 27
The length of an ECS buffer is not evenly divisible by the length of an in-core buffer. The program continues.

ILLEGAL HASH PAGE NØ - 15
File directory page number either LE 0 or GT ISSIZE

ILLEGAL ORDINAL NO. ON DELETE - 10
Data on disk bad, usually occurs on call from ADHASH or RPHASH.

ILLEGAL ORDINAL NO. ON ENTER - 7
Data on disk bad, usually occurs on call of ADHASH or RPHASH

ILLEGAL ORDINAL NO. ON GET - 9.
Data on disk bad or bad parameters in GETHSH. Perhaps a record's address was expected to be returned from GETLOC, but actually a hash address was returned - IFLAG should be checked.

ILLEGAL ORDINAL NO. ON REPLACE - 8
Data on disk bad, usually occurs on call to RPHASH.

ILLEGAL PAGE NUMBER - 5
Bad data on disk or incorrect arguments input to GETHSH

ILLEGAL PAGE SIZE - 4
On a call to INIT or INIT1, the number of sectors per page was LT 0 or GT 28; program terminates with STOP 4.

ISSIZE NEGATIVE - 3
ISSIZE negative on call to INIT or INIT1; program terminates with STOP 3.

LABELED COMMON TOO SMALL - 1
Size of labeled common specified is too small for number and size of I/O slots requested; program terminates with STOP 1 if error occurs in FILSLT. If the error occurs in FILECS - not enough room for NOSLT headers - program continues.

LENGTHS NOT EQUAL ON REPLACE - 12
Bad data on disk.

NAME OF REPLACING ENTITY DIFFERENT - 19

Name of entity to replace, in RPHASH, is different from record already on disk with address in IPNTR(1).

NOT ENOUGH ROOM ON ENTER - 11

Bad data on disk or program writing over labeled common area.

POINTER BUFFER TOO SMALL - 2

Pointer buffer size - as specified in FILSLT or SETSZ - is less than (NPOINT+1)*3; program terminates with STOP 2.

SET SAME PARENT TWICE - 21

Attempt to add same entity to a given linked list more than one time.

TOO MANY GENERATIONS - 25

More than seventeen generations have been used - possibly circular tree. Message appears from SCOOP or SAVE. SCOOP continues with next primary parent.

TOO MANY PARENTS - 20

Maximum number of parents exceeded.

TYPE OUT OF RANGE - 22

Pointer type either LT 0 or GT 2**11-1. Only bottom eleven bits used.

LIST OF FIGURES

INDEX TO SUBROUTINES