CONTROL DATA
CORPORATION

CONTROL DATA®
1700 COMPUTER SYSTEMS

1700 MSOS 4 FILE MANAGER VERSION 1 SOFTWARE REFERENCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A	Manual Released
(4/74)	
Publication No. 39520600	

Additional copies of this manual may be obtained from the nearest Control Data Corporation sales office.

Address comments concerning this manual to:

Control Data Corporation
Small Computer Development Division
4455 Eastgate Mall
La Jolla, California 92037

or use Comment Sheet in the back of this manual.

CONTENTS

FIGURES

TABLES

INTRODUCTION 1

The 1700 File Manager is a general-purpose file management package consisting of a request supervisor and a collection of request processors. The supervisor resides in core and the request processors on mass storage; core requirements are minimized by bringing in individual request processors only as they are needed.

The File Manager creates and maintains either sequential or indexed files. The records in any file may be variable in length and may be added, replaced, or removed at any time after the file has been defined and before it is released.

A sequential file is one in which each new record is added immediately following the last record stored in the file. These records must be retrieved in the same sequence in which they were stored and cannot be retrieved at random. Thus, they are retrieved on a FIFO basis.

An indexed file is one in which each record has an identifier or key (surname, social security number, etc.). These records are stored sequentially with a key value; records with the same key value may be linked together. These records may be retrieved sequentially or a specific record may be retrieved by using its key value.

GENERAL FILE FEATURES 2

2.1 STORAGE AND RETRIEVAL

The File Manager stores and retrieves information in three basic ways:

- Sequential
- Indexed
- Direct

In addition, variations for storage and retrieval are provided by combinations of the preceding and special options. The variations are:

- Indexed-ordered
- Indexed-linked

2.1.1 SEQUENTIAL

In sequential access, records are stored one at a time immediately following the last record stored and are retrieved one at a time in the same order they were stored, starting from the beginning of the file.

Sequential access is best suited for retrieving all records on a FIFO basis. It is not suited to retrieving a particular record since all preceding records must first be retrieved.

NOTE

All files may be accessed sequentially.

2.1.2 INDEXED

Indexed access is best suited for access of a specific record. Each record may be indexed by one and only one key. The key may be one or more words in length. Since all files are sequential, an indexed file may be termed indexed-sequential. Indexed access is only possible from an indexed file.

A particular record can be stored and retrieved via a key. Each record key value can be translated into an index which can provide relatively quick access to the record. Indexed files require extra file space for the keys and key directories.

2.1.3 DIRECT

Direct access is best suited for frequently accessed records. A record must have been stored in a sequential or indexed file by a sequential or indexed store before it can be referenced directly.

Direct procedures are normally used in updating records and in forming list structures. Since the File Manager provides record pointers for all records, all the files may be accessed directly.

2.1.4 VARIATIONS

2.1.4.1 INDEXED-ORDERED

When the indexed-ordered option is selected, indexed records can be retrieved in a manner similar to sequential retrieval. However, instead of a FIFO basis, records are retrieved starting at the record with the lowest numeric key value (or the key value specified in the first of the repeated index-ordered retrieve) and continuing through to the record with the highest numeric key value. When this type of access is used, a sort of the key values is done; therefore, the key value must be one word in length. It is recommended that key values for an indexed-ordered file include only non-negative values.

2.1.4.2 INDEXED-LINKED

In an indexed file, each record normally has a unique key value. However, if the indexed-linked option is selected, records with the same key value are linked together in either a LIFO or FIFO manner. The records are linked by allocating two words of each record for the linking record pointer. The retrieval of these records is an example of a list structure and is described in detail in Section 3.3. LIFO or FIFO linking is specified by the user when a file is defined as indexed.

2.1.4.3 LIST STRUCTURES

Records may be retrieved as though they were part of a list structure by using the record pointers supplied by the File Manager and the direct method of retrieval. The user may form complex list structures by linking forward, backward, ring, sublist, etc. A record may be a member of an indefinite number of lists as long as two words for a record pointer are reserved in the record for each list. An example of a list structure is an indexed-linked file.

2.2 FILE REQUEST

The four types of file requests are described in Section 3. They are:

- Specification — Specification requests provide for:

  -Defining a file

  -Defining an indexed file

  -Locking a file

  -Unlocking a file

  -Releasing a file

- Sequential
- Indexed
- Direct

} These three file requests are used to store and retrieve information.

The file manager executes a request at the caller's priority level. If, however, the File Manager is executing a previous request, the request is queued by its priority level and is not executed until the currently active request and any higher level waiting requests have been executed.

Associated with each request are a 12-word temporary buffer and one indicator word. The buffer is used to process the file request; the indicator word denotes the status of the file request upon completion. Each bit of the indicator word which is non-zero signifies an abnormal occurrence. If bit 15 is non-zero, the request is rejected because of errors denoted in the other bits; if bit 15 is a zero but other bits are non-zero, the request is completed with an irregular occurrence (for example, an end-of-file has occurred). Note that bit 14 is a common bit for rejecting a request due to invalid parameters in a request. If the entire indicator word is zero, the request terminates normally.

2.3 RECORD FORMAT

Each variable-length record is composed of three sections: header word, record pointers, and data words.

HEADER WORD

The first word of each record is reserved exclusively for the header word. The File Manager sets this word to the total length of the record when this record is stored. Once a record is defined, its length (and consequently the header word) cannot be changed.

NOTE

When storing and retrieving a record, the number of words in a record must include the header word.

RECORD POINTERS

A record in an indexed-linked file includes a record pointer. A record pointer is a two-word mass-storage address which points to another record on mass storage. The first word is the sector location of the file record block in which this record resides. The second word contains the word the record starts in. If a file is indexed-linked, the second and third words are reserved for the record pointer, which points to the last record that was stored with the same key value. This is the same format as the recptr parameter passed back to the user from STOSEQ and STOIDX requests (see Sections 3.2.1 and 3.3.1).

DATA WORDS

Each record may have zero or more data words, which contain the actual record information. The information may be binary or ASCII.

2.4 UPDATE PROTECTION

Whenever a record is to be updated, the user must retrieve the record and lock the file with a unique file combination, subsequently storing the updated record and unlocking the file with the same file combination, utilizing the store direct request (refer to Section 3.4). More than one record may be retrieved, updated, and restored as long as the same file combination that was used to lock the file is supplied. Note that the file should not be locked for an extended period of time because other users, who may also wish to update, cannot access the file until it is unlocked. Thus a retrieve, which attempts to lock an already locked file with a different combination, is queued and cannot be executed until the file is unlocked.

If a number of files are to be locked, it is advisable to lock and unlock the files in a given sequence. For example, lock files in ascending numerical order and unlock them in descending numerical order.

A retrieve without a file combination or a store of a new record is permitted on a locked file with the understanding that one or more records of that file are in the process of being updated. Note that an update into an unlocked file or a locked file using an incorrect file combination results in a file request error.

The file combination must necessarily be unique so that no two requests use the same file combination. This can be accomplished by using the ASSIGN statement in FORTRAN or the RTJ instruction in Assembly language.

2.5 UNPROTECTED FILE REQUESTS

Unprotected programs are assumed not to be error-free; therefore, certain restrictions have been placed on unprotected file requests.

An unprotected file request cannot update a record in a file because it cannot use the store direct request. This restriction is imposed because the File Manager has no way to check the validity of the record pointer in the store direct request. The restriction is mitigated by the assumption that background programs primarily retrieve records (for example, data reduction, analysis, etc.) and that records can always be retrieved, updated, and stored as new records in another unprotected file.

Since updates cannot be done, file locking is illegal for unprotected file requests. Note that unprotected file requests may not store records into or remove records from files that were defined by protected programs.

NOTE

If there is not enough allocatable core for both the File Manager and Job Processor modules, file requests from background can hang batch processing indefinitely.

2.6 REQUIREMENTS AND LIMITATIONS

The File Manager requires certain information to establish the file structure and imposes limitations on those files.

2.6.1 MAXIMUM RECORD LENGTH

The effective maximum record length and file record block length are determined as a function of the maximum record length specified by the define file request for each file. It places a maximum limit on the length of records for that file, and also establishes a block of sector(s) that will be allocated when the first record is stored into the file. Subsequent records are stored into this block until it is full, then another equal block of sector(s) is automatically allocated. This process is continued as long as there is mass memory space available. Thus a file record block may contain one or more records (see Appendix B, especially Section B.2.2).

The effective maximum record length is equal to the specified maximum record length if the specified maximum record length plus 3 is equal to an integral multiple of 96. Otherwise, the effective maximum record length is equal to the least integer value n such that

1) n is greater than the specified maximum record length, and
2) $N = 96 \cdot m - 3$ for some positive integer m.

Thus, specified maximum record length values of 3, 93, and 94 would result in effective maximum record lengths of 93, 93, and 189 respectively.

2.6.2 EXPECTED NUMBER OF RECORDS WITH DIFFERENT KEY VALUES

The expected number of records with different key values is specified by the define file indexed (DEFIDX) request parameter numekv (refer to Section 3.1) for each indexed file. Note that if a file is not indexed-linked, this is equivalent to the number of records in the file. The expected number of records with different key values establishes the structure of the indexed directories. A relatively accurate estimate is important if the number of expected key values exceeds 8,464.

Too low an estimate may result in more mass storage accesses per indexed request, while too high an estimate may result in excessive core allocation for the indexed directories per indexed request. (Refer to Section A.4.4.)

2.6.3 PARAMETER LIMITATIONS

The following limitations are necessary:

File number range	1 through 32,767
Record length range	1 through 32,767
Number of expected records (with different key values) range	1 through 32,767
Key value length range	1 to 63 words
File combination range	1 through 32,767
Key value range for indexed-ordered files	0 through 32,767

CAUTION

Users are warned that programs making File Manager requests which contain relative parameters will not execute properly in partitioned core or at addresses above 8000_{16}.

2.6.4 RESTRICTIONS WHEN USING MORE THAN ONE LOGICAL UNIT FOR FILE SPACE

As noted in Section 2.7.2, the File Manager initializes the file space pool for every File Manager logical unit at the same time. This initialization occurs when the first define file request is encountered after a 1700 operating system is built. The initialization of a file space pool for a given logical unit involves writing the file space pool thread on that unit in the first sector which is available for File Manager use. This procedure necessitates the following warnings to the user.

CAUTION

1. At the time the first define file request is executed, all logical units which are to be used, now or later, for file storage must be operating.
2. The disk pack mounted on a drive at the time the first define file request is executed must be mounted on that drive when the File Manager uses that unit for storage.

2.7 SPACE ALLOCATION

2.7.1 FILE SPACE ALLOCATION

File space is allocated by the define file request for file records or by the define file indexed request for file indexed directories. Provision is made to return file space by the release file request and the retrieve/remove requests. Note that:

- All a file's records must be on one logical unit.
- All a file's indexed directories must be on one logical unit.
- Only logical units which are mass memory devices are currently allowed.

The first time a define file request is encountered after a 1700 operating system with the File Manager is built, a file space list and a file space pool are constructed for each logical unit that has available file space. The File Manager tests the value of the SYSDAT FILMGR core location FIDSEC. The value of FIDSEC is zero until after the first define file request is encountered.

A file space list is composed of one or more blocks of available space. Each block is a threaded sequence of segments of mass memory such that each segment has the same length in sectors as every other segment in the block.

For example, there may be a block of all available two-sector segments. At the time the system is built, the user determines what blocks (i.e., what available segment lengths) are to be included in the file space list. All available file space which lies in a segment of a length other than those included in the file space list is included in a file space pool. The advantage of keeping as much of the available file space as possible in the file space list is that available space there can be allocated much more quickly than can space in the file space pool. The disadvantage of having too many blocks in a file space list is that two words of core (in SYSDAT) are used for each block.

Consider the following example. The user determines that the file space list is to be composed of

- A block of segments one sector long,
- A block of segments two sectors long, and
- A block of segments four sectors long.

Suppose at a given time there are file space segments of various lengths: one sector, two sectors, three sectors, four sectors, six sectors, eight sectors, 20 sectors, and 10,000 sectors. The file space list and the file space pool are represented in Figure 2-1.

The efficiency of the File Manager can be optimized by determining, at the time the system is built, what file space lengths will be used. If only a small number of different lengths are needed, a block for each of these lengths can be included in the file space list. For example, suppose only segments of one sector, two sectors, and four sectors are to be allocated for file space. A block for each of these lengths in the file space list requires only six words of core storage in SYSDAT.

Refer to Appendix A for details concerning file space requirements. Remember that space must be allocated for key directories and key information segment blocks as well as for file records when using indexed files.

2.7.2 FILE SPACE AUDIT

When there is insufficient file space to define a new file or to store another record in a given file, the File Manager will indicate this condition to the requestor. In many cases, it is desirable to know when file space is running low before all the file space is gone. Files which are no longer in use could then be released to make additional space. A user early-warning program may be written to monitor the ratio of available space and total space for each logical unit. These parameters are located in the FILMGR SYSDAT parameter area. For example, for File Manager logical unit 1 we may find in SYSDAT:

```
LUE1    NUM
        NUM
        NUM    x      AVAILABLE FILE SPACE
        NUM    y      TOTAL FILE SPACE
```

Thus, one could calculate for logical unit one:

$$P_1 = \frac{(\text{LUE1} + 2)}{(\text{LUE1} + 3)} = \frac{x}{y}$$

giving the ratio of file space available to original file space for this logical unit. Location LUE1 is the same location as that of FSLIST in Figure 2-1.

2.7.3 CORE ALLOCATION

The individual file request processors (e.g., store sequential), the information segment for a file (FIS — see Section A.2), and its indexed directory (KIS — see Section A.4) are placed in allocated core. Each item will remain in core to conserve mass memory accesses, as long as it enjoys sufficient usage by the File Manager user(s). Once a certain item has not been utilized for a period of time, its core will be released and the next use of it will require a mass memory transfer.

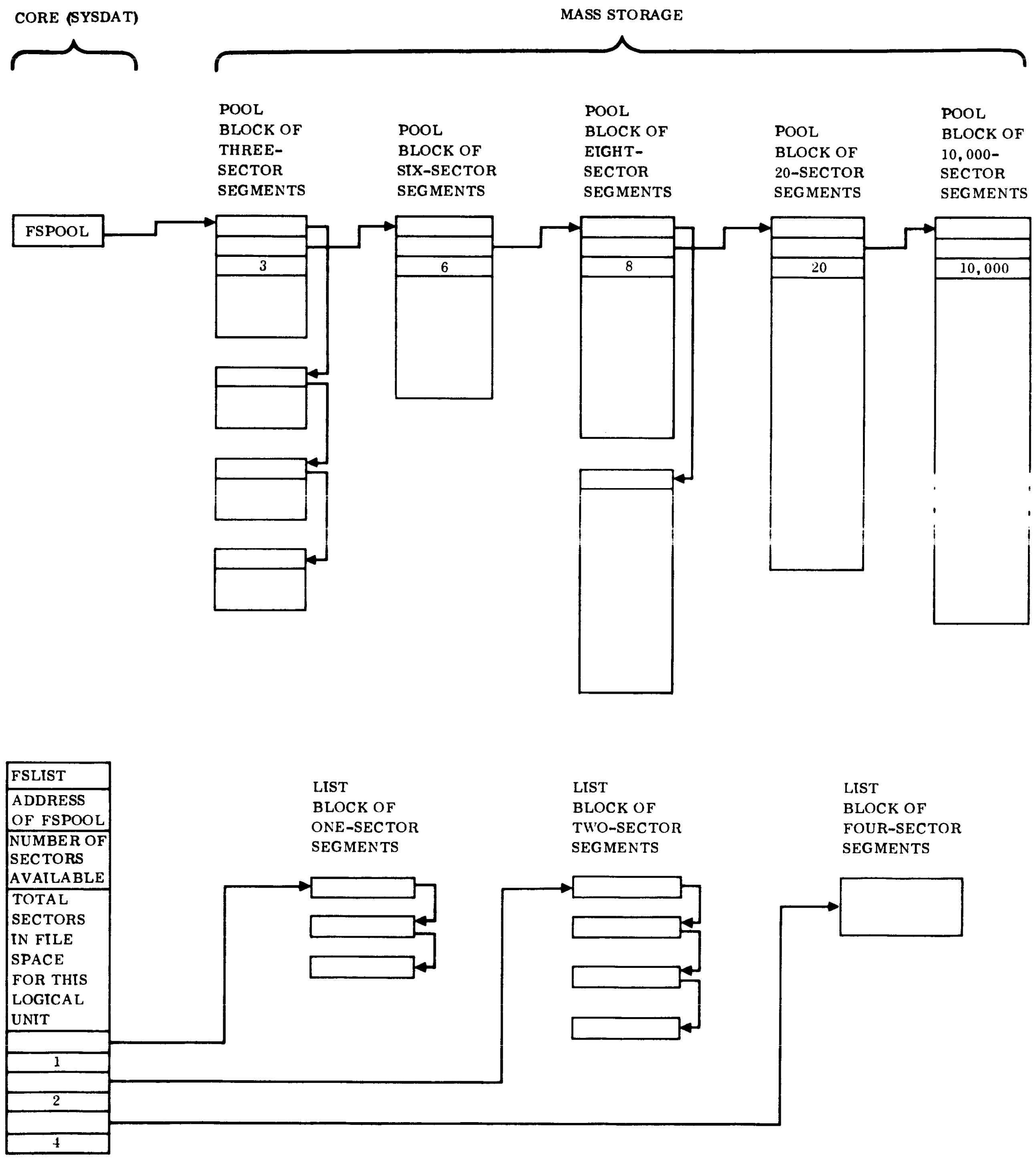

CORE (SYSDAT)
MASS STORAGE
FSPOOL
POOL BLOCK OF THREE-SECTOR SEGMENTS
3
POOL BLOCK OF SIX-SECTOR SEGMENTS
6
POOL BLOCK OF EIGHT-SECTOR SEGMENTS
8
POOL BLOCK OF 20-SECTOR SEGMENTS
20
POOL BLOCK OF 10,000-SECTOR SEGMENTS
10,000
FSLIST
ADDRESS OF FSPOOL
NUMBER OF SECTORS AVAILABLE
TOTAL SECTORS IN FILE SPACE FOR THIS LOGICAL UNIT
1
2
4
LIST BLOCK OF ONE-SECTOR SEGMENTS
LIST BLOCK OF TWO-SECTOR SEGMENTS
LIST BLOCK OF FOUR-SECTOR SEGMENTS

Figure 2-1. Example of File Space Pool and File Space Test

The time period for each of the above is a system parameter determined when the 1700 operating system is built.

2.8 FILE VALIDITY CHECK

To minimize mass memory I/O traffic, the File Manager allows file information to remain in allocatable core until a time-out occurs, at which time the information is updated on mass storage.

CAUTION

Abnormal system stops and autoloads can destroy this information and will eventually cause fatal file errors.

If the system contains a File Manager, a file validity check is performed each time the system is autoloaded. The check is preceded by the message:

```
CHECKING FILES -
```

on the system comment device, and consists of a trace of all file spacė threads on mass storage. If the threads are found to be valid an OK is printed. If any errors are found the user is given the option of continuing with the autoload or purging all system files (i.e., reverting all File Manager tables to a condition prior to the loading of any files). If this option is selected the files would have to be reloaded from a user-written backup dump.

FILE REQUEST DESCRIPTIONS AND CALLS 3

All file request calls to the File Manager may be written as FORTRAN type calls or as Assembly language macros as in the following descriptions. The Assembly language call format, without the use of macros is described in Section 3.5.

CAUTION

Programs making File Manager requests which contain relative parameters will not execute properly in partitioned core or at addresses above 8000_{16}.

3.1 SPECIFICATION REQUESTS

Specification descriptions and calls are discussed in the following order:

- Define file
- Define file indexed
- Lock file
- Unlock file
- Release file

3.1.1 DEFINE FILE

A file must be defined before any information can be stored or retrieved. A file cannot be defined if it is already defined. However, the file could be redefined if it had been previously released (refer to release file in this section).

The define file request specifies:

- The file number of the file being defined (permitting other requests to reference this file)
- A value to determine the length in words of each block of file space which is allocated to a file when needed. This value also places an upper limit on the length of any record in a file. Any attempt to exceed this limit results in an error (refer to Section 2.6).
- A logical unit where the file's records will be stored

- A temporary buffer for processing the request
- An indicator word, denoting the request's status upon completion

The FORTRAN format for the define file call is as follows.

```
CALL DEFFIL (filnum, maxrl,lu,reqbuf,reqind)
```

Where:	filnum	is	the file number; it contains a positive integer specifying the file to be defined.
	maxrl	is	the maximum record length; to be used for determining the effective maximum record length and file record block length. It contains a positive integer. See Sections 2.6.1 and 3.1.1.
	lu	is	the logical unit; it contains a positive integer specifying where the file's records are to be stored.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (see below).

The Assembly language macro format is as follows:

DEFFIL	filnum	(Produces a call with an absolute address for filnum)
DEFFIL*	filnum	(Produces a call with a relative address for filnum, as for a run-anywhere program)

The parameter filnum is defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3.5.2.

The following is a list of the file request indicator bits. This list is referenced for the reqind parameter in each of the request calls.

0	File defined/not defined
1	File locked/not locked
2	Long store or short read
3	End-of-file encountered
4	At least one more record exists with the same key value
5	Record does not exist or has been removed
6	Unused
7	Mass storage error
8	No more file space left
9	Attempt to store direct outside File Manager's disk space
10	File combination incorrect

11	File already defined/not defined as indexed
12	Key length not one for indexed-ordered file
13	Unprotected file request attempt to change a protected file
14	File request illegal
15	File request rejected; this bit is set whenever

Bits 14, 13, 12, 11, 10, 8, 7, 5, or 0 are set

Bit 4 is set for STOIDX if the file has not been defined as linked

Bit 2 is set for STOSEQ/STOIDX

Bit 1 is set for RELFIL, UNLFIL, STODIR, LOKFIL (already locked), RTVSEQ, RTVIDX, RTVIDO, and RTVDIR (attempt to remove from locked file without the combination).

An Assembly language macro to test specified bits of the request indicator word is described in Section 3.5.3.

3.1.2 DEFINE FILE INDEXED

Define indexed is used to further define, as indexed, those files which have already been defined by a call to DEFFIL. A file must be defined indexed before any information can be stored or retrieved via an indexed key. Normally, if a file is to be indexed, this request is made immediately after it is defined.

If the records of a file are to be ordered by key value, the key length of the record must be one word. A file cannot be defined indexed if it is not defined, if it is already defined indexed, or if records have already been stored sequentially into it. An unprotected program cannot define indexed a file which was defined by a protected program.

The FORTRAN format for the define file indexed call is as follows:

```
CALL DEFIDX (filnum, numekv, keylth, lu, reqbuf, reqind)
```

Where:	filnum	is	the file number; it contains a positive integer specifying the file to be defined as indexed.
	numekv	is	the number of expected key values; it contains a positive integer estimating the number of records with different key values to be stored in the file.

keylth	is	the key length word, with the indexed options:

Bits 0 through 5	Length of the key	
6 through 12	Reserved	
13	1	FIFO linking (bit 15 must be set). If this bit is not set and bit 15 is set, LIFO linking is implied.
14	1	Indexed-ordered file
15	1	Indexed-linked file

lu	is	the logical unit; it contains a positive integer specifying where the file's indexed directories are to be stored.
reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process a request.
reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (see reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

DEFIDX	filnum	(Produces code with an absolute parameter address)
DEFIDX*	filnum	(Produces a relative parameter address as for a run-anywhere program)

The parameter filnum is defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3.5.2.

3.1.3 LOCK FILE FOR PROTECTED PROGRAMS ONLY

A file may be locked by protected programs when it is possible that more than one program may be attempting to update the same file.

A file cannot be locked if it is not defined, if it is already locked, or if the lock file request is issued on a protected file by an unprotected program. An alternate method of locking a file is provided in the retrieve request (refer to Sections 3.2, 3.3, and 3.4).

The lock file request specifies:

- The file number of the file being locked
- The file combination required to store in this file
- A temporary buffer for processing this request
- An indicator word, denoting request's status upon completion

The FORTRAN format for lock file call is as follows:

```
     CALL LOKFIL (filnum,filcom,reqbuf,reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file being locked.
	filcom	is	the file combination with the remove option; bits 0 through 14 contain a non-zero number which must be used in subsequent store or remove requests; bit 15 is not used.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

LOKFIL	filnum	(Produces a call with an absolute address for filnum)
LOKFIL*	filnum	(Produces a call with a relative address for filnum, as for a run-anywhere program)

The parameter filnum is defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program as described in Section 3.5.2.

3.1.4 UNLOCK FILE FOR PROTECTED PROGRAMS ONLY

A file may be unlocked when there are no further updates to be done. The same file combination that was used to lock the file must be used to unlock it. An alternate method of unlocking a file is provided in the store direct request (refer to Section 3.4).

A file cannot be unlocked if it is not defined or not locked, if the combination is incorrect, or if an unprotected program attempts to unlock a file defined by a protected program.

The unlock file request specifies:

- The file number of the file being unlocked
- The file combination used to previously lock the file
- A temporary buffer for processing the request
- An indicator word, denoting the request's status upon completion

The FORTRAN format for unlock file call is as follows:

CALL UNLFIL (filnum, filcom, reqbuf, reqind)

Where:	filnum	is	the file number; it contains a positive integer identifying the file being unlocked.
	filcom	is	the file combination; bits 0 through 14 contain a non-zero number which is identical to the combination that was used to lock the file; bit 15 is not used.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3. 1).

The Assembly language macro format is as follows:

UNLFIL	filnum	(Produces a call with an absolute address for filnum)
UNLFIL*	filnum	(Produces a call with a relative address for filnum, as for a run-anywhere program)

The parameter filnum is defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program as described in Section 3. 5. 2.

3.1.5 RELEASE FILE

A file may be released when there is no further use for it. This results in all the space reserved for its data records, and any information associated with indexing, being returned for future utilization by other files.

A file cannot be released if it is not defined or if it is locked. An unprotected program cannot release a file defined by a protected program.

The release file request specifies:

- The file number of the file being released
- A temporary buffer for processing the request
- An indicator word, denoting the request's status upon completion

The FORTRAN format for release file call is as follows:

CALL RELFIL (filnum, reqbuf, reqind)

Where:	filnum	is	the file number; it contains a positive integer specifying the file to be released.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

RELFIL	filnum	(Produces a call with an absolute address for filnum)
RELFIL*	filnum	(Produces a call with a relative address for filnum, as for a run-anywhere program)

The parameter filnum is defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program as described in Section 3.5.2.

3.1.6 EXAMPLES OF SPECIFICATIONS REQUESTS

As a part of a computer system to aid a law enforcement agency, a first offense file is defined. The record format is to be as follows:

Word	Contents
1	Header
2-25	Name
26-28	Date of first arrest
29-30	Time of first arrest
31	Violation code

Note that a record length of 31, a factor of 93, is used. By setting the maximum record length to 93, the maximum record length and the effective maximum record length are equal. Thus, no space is wasted in the mass memory storage of the file records.

Periodically, the contents of the first offense file are to be written on magnetic tape. The FORTRAN code shown in Example 1 is used to define the file initially as well as to re-initialize the file after each transfer of its contents to tape.

Note that a release file request occurs before the file is defined. Therefore, when the file is defined it will contain none of the previously stored records.

The file records are to be accessed by license number, expressed as four ASCII words. Note that the license number is not part of the data in the record, but that its length in words is specified as the key length when the file is defined as indexed. An estimate of 1,000 first offenses will be recorded in the file before the file is cleared and re-defined. Thus, NUMEKV = 1000.

EXAMPLE 1:

```
        INTEGER REQBUF{12},REQIND
        DIMENSION IOBUF{58}
        .    .         .   .
        .    .         .   .
        .    .         .   .
    C RELEASE THE FIRST OFFENSE FILE

        IFLNUM=20

        CALL RELFIL{IFLNUM,REQBUF,REQIND}

    C CHECK FOR ERRORS

        IF {REQIND.LT.0} GO TO 1000

    C DEFINE THE FIRST OFFENSE FILE

        MAXRL=93

    C LOAD A-REGISTER WITH FILES LOGICAL UNIT, IFLU, AN EXTERNAL.
    C SAVE IN LOCAL VARIABLE, LU.
    C      LDA  =XIFLU
    C      STA  LU

        ASSEM $C000,+IFLU,$6800,LU

        CALL DEFFIL {IFLNUM,MAXRL,LU,REQBUF,REQIND}
    C CHECK FOR ERRORS
        IF {REQIND.LT.0} GO TO 1000
    C DEFINE FILE TO BE INDEXED BY LICENSE NUMBER
        KEYLTH=4
        NUMEKV=1000
        CALL DEFIDX {IFLNUM,NUMEKV,KEYLTH,LU,REQBUF,REQIND}

        IF {REQIND.LT.0} GO TO 1000
        .    .         .   .
        .    .         .   .
        .    .         .   .
     1000 CALL SETBFR{IOBUF,58}
        WRITE {4,3000} IFLNUM,REQIND
    C FOLLOW ERROR EXIT PATH
        .    .         .   .
        .    .         .   .
        .    .         .   .

     3000 FORMAT {5HFILE ,I5,8H ERROR $,$4}
```

The FORTRAN code shown in Example 2 illustrates the proper method for determining a file combination so that uniqueness is guaranteed. The figure also shows acceptable call statements to lock and unlock a file. This code must be part of a protected program according to File Manager restrictions.

<u>EXAMPLE 2:</u>

```
      INTEGER REQBUF{12},REQIND,FILCOM
      DIMENSION IOBUF{58}.
        .                  .
        .                  .
        .                  .
      IFLNUM=IBASE+303
C NOTE**ONLY THIS METHOD SHOULD BE USED TO GENERATE FILE
C COMBINATIONS**
 2000 ASSIGN 2000 TO FILCOM



      CALL LOKFIL{IFLNUM,FILCOM,REQBUF,REQIND}
      IF   {REQIND.LT.0} GO TO 1000
        .                  .
        .                  .
        .                  .
C UPDATE FILE
        .                  .
        .                  .
        .                  .
      CALL UNLFIL{IFLNUM,FILCOM,REQBUF,REQIND}
      IF   {REQIND.LT.0} GO TO 1000



 1000 CALL SETBFR{IOBUF,58}
      WRITE {4,3000} IFLNUM,REQIND
C FOLLOW ERROR EXIT PATH
        .                  .
        .                  .
        .                  .
 3000 FORMAT {5HFILE,I5,8H ERROR $,$4}
```

The FORTRAN code for a part of the files initialization procedure for a given system is shown in Example 3. The first define file request is used as a test to see if the file has been previously defined. If so, the information in the file will be used before the file is released and re-defined. If not, initial condition records are stored in the file.

EXAMPLE 3:

```
      DIMENSION IREQBF(12),IOBUF(58)
        .                 .
        .                 .
        .                 .
      IFLNUM=5
      MAXRL=93
      LU=8
      CALL DEFFIL(IFLNUM,MAXRL,LU,IREQBF,IREQID)
C WAS FILE PREVIOUSLY UNDEFINED
      IF (AND(IREQID,1).EQ.0) GO TO 1000
C FILE WAS PREVIOUSLY DEFINED
C CHECK FOR FILE ERRORS
      IF (AND(IREQID,$0418).NE.0) GO TO 5000
C NO FILE ERRORS, PROCEED TO USE
C INFORMATION IN FILE
        .                 .
        .                 .
        .                 .
C OLD FILE INFORMATION HAS BEEN USED.
C RELEASE AND RE-DEFINE FILE 5
      CALL RELFIL(IFLNUM,IREQBF,IREQID)
      IF (IREQID.LT.0) GO TO 5000
      CALL DEFFIL (IFLNUM,MAXRL,LU,IREQBF,IREQID)
      IF (IREQID.LT.0) GO TO 5000
C STORE INITIAL CONDITIONS RECORDS INTO FILE
 1000   .                 .
        .                 .
        .                 .
C PRINT ERROR MESSAGE
 5000 CALL SETBFR(IOBUF,58)
      WRITE (4,6000) IFLNUM,IREQID
 6000 FORMAT (6H FILE, I5,8H ERROR $,$4)
```

3.2 SEQUENTIAL REQUESTS

3.2.1 STORE SEQUENTIAL RECORD

Records may be stored sequentially in a file once it is defined. A record is always stored as the last record of the file, with its record pointer returned to the caller so that the record may be accessed directly (refer to Section 3.4). A sequential store is permitted in a locked file with an indication being given that the file was locked.

The length of the record cannot exceed the specified maximum record length (refer to Section 2.6). A record cannot be stored sequentially if the file is indexed or if it is not defined. An unprotected program cannot store a record in a file which is defined by a protected program.

The store sequential record request specifies:

- The file number of the file where the record is being stored
- A buffer for returning the record pointer
- A buffer of information to be stored as the record
- A temporary buffer for processing the request
- An indicator word, denoting the request's status upon completion

The format for store sequential record call is as follows:

```
CALL STOSEQ (filnum, recptr, recbuf, reclth, reqbuf, reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file into which a record is being stored.
	recptr	is	the record pointer; it is a two-word array, set by the File Manager, which contains the record pointer of where the record was stored.
	recbuf	is	the record buffer; it is an array of reclth words containing the record to be stored.
	reclth	is	the record length; it contains a positive integer specifying the length of the record.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

STOSEQ	filnum, recbuf, reclth	(Produces a call with an absolute address for each parameter)
STOSEQ*	filnum, recbuf, reclth	(Produces a call with a relative address for each parameter, as for a run-anywhere program)

The parameters filnum, recbuf, and reclth are defined as in the FORTRAN call. If the reclth parameter is blank, the current record length is left unchanged. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3.5.2.

CAUTION

The File Manager assumes that the three words preceding the record buffer are part of the user's program. To optimize storing time, the contents of these three words are temporarily altered to contain the FRB header words whenever the File Manager writes the first record in an FRB to mass memory (refer to A.3). After the transfer occurs, the File Manager restores the original contents of the three words.

The user's program may be in a multiprogramming environment. In this case, if the record buffer is at the beginning of the program and the three words preceding the record buffer extend beyond the user's program, the three words could overlap another program or core storage, such as a core allocation thread which might be used before the original contents are restored.

3.2.2 RETRIEVE SEQUENTIAL RECORD

Record(s) may be retrieved sequentially from a file once it is defined and at least one record has been previously stored into the file. (The file may have been defined by either a protected or an unprotected program.) Each record in the file may be retrieved sequentially by repeatedly executing one RTVSEQ call (see Section 3.2.2) until an end-of-file indication is given. If there are n records in a file, the first RTVSEQ call will retrieve the first record that was stored into the file, the nth RTVSEQ call will retrieve the last record that was stored into the file, and the n + 1st RTVSEQ call will produce an end-of-file indication. For each of the n calls, one record is retrieved along with its corresponding record pointer, so that the record may be accessed directly (see Section 3.4). If there are no records in the file, the first call will produce an end-of-file indication.

General information for retrieve sequential records is as follows:

- It is not necessary to retrieve all the records from the file.
- The first and subsequent records can be re-retrieved by a new call or by re-initializing the current call record pointer.

CAUTION

Parameters of a repeated RTVSEQ call cannot be altered between calls.

For update purposes, a file may be locked with a file combination as a record is retrieved or a record may be retrieved from a previously locked file (see Section 3.1.3) if no file combination is specified or if the same file combination that was used to lock the file is specified. These retrieved and updated record(s) may be restored into the locked file via the store direct record call (see Section 3.4.1), again only if the same file combination is used. Note that a sequential retrieve without a file combination is permitted from a locked file with an indication being given by the File Manager that the file was locked. Provision is also made for removing a record from a non-indexed file as it is retrieved. The first part of a record of any desired length may be retrieved with an indication made that there was a short retrieve.

A record may be retrieved, but cannot be removed from an indexed file using a retrieve sequential request. A record is not retrieved sequentially if the record was previously removed from the file (however, subsequent records which exist may be retrieved by repeating the same call). A record cannot be retrieved if the file is not defined. An unprotected program may retrieve, but cannot remove, a record from a file which was defined by a protected program.

The retrieve sequential record request specifies:

- The file number of the file from which the record is being retrieved
- The file combination, if the file is to be locked, or the record that is to be retrieved from a locked file
- Whether the record is to be removed from the file
- A buffer for returning the record pointer
- A buffer for receiving the record to be retrieved
- A temporary buffer for processing the request
- An indicator word, denoting the request's status upon completion.

The format for retrieve sequential record call is as follows:

```
CALL RTVSEQ (filnum,filcom,recptr,recbuf,reclth,reqbuf,reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file from which the record is to be retrieved.
	filcom	is	the file combination with the remove option: bits 0 through 14 contain a non-zero number (if the file is or is to be locked), specifying the combination (which is or is to be) used to lock the file; bit 15 set to a one indicates that the record is to be removed from the file.
	recptr	is	the record pointer; it is a two-word array set by the File Manager, which contains the record pointer from where the record was retrieved. Initially, both words must be set to zero by the requester.
	recbuf	is	the record buffer; it is a non-preset array of reclth words, where the File Manager transfers the retrieved record.
	reclth	is	the record buffer length; it contains a positive integer specifying the length of the record buffer.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3. 1).

The Assembly language macro format is as follows:

RTVSEQ	filnum, recbuf	(Produces a call with an absolute address for each parameter)
RTVSEQ*	filnum, recbuf	(Produces a call with a relative address for each parameter, as for a run-anywhere program)

The parameters filnum and recbuf are defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3. 5. 2.

3.2.3 EXAMPLES OF SEQUENTIAL REQUESTS

A computerized supermarket stores the orders of its customers in the new orders file, sequentially in the order in which the orders are phoned in. The FORTRAN code in Example 4 is part of the code executed each time a new order is entered into the computer system. Note that the record data is stored beginning at word 2 of the record to allow for the header word.

EXAMPLE 4:

```
      COMMON INPBUF(284)
      DIMENSION IRECPT(2),IREQBF(12),IRECBF(285)
      DIMENSION IDATA(284)
      EQUIVALENCE (IDATA(1),IRECBF(2))
C TRANSFER CUSTOMER NAME,LOCATION CODE,
C ROUTE NUMBER, AND ITEMS ORDERED FROM INPUT
C BUFFER TO RECORD BUFFER
      DO 100 I=1,284
  100 IDATA(I)=INPBUF(I)
C STORE RECORD INTO NEW ORDERS FILE
      IFLNUM=5
      CALL STOSEQ(IFLNUM,IRECPT,IRECBF,285,IREQBF,IREQID)
C CHECK FOR ERRORS
      IF (IREQID.LT.0) GO TO 9000
```

In the next example the new orders file described in Example 1 is processed to list all orders to be delivered on a given route. The records in the file are retrieved sequentially. When a record is found which corresponds to the given delivery route, the record is removed from the new orders file and stored in the routed orders file.

The tests for end-of-file and previous record removal appear before any checks on the contents of the record buffer, since it must first be ascertained that a record was actually retrieved. Records are removed via direct retrieval, using the record pointer returned from the File Manager in the sequential retrieve call. Note that the remove option code, bit 15 of the third parameter in the retrieve direct calling list, is set to one to indicate that record removal is requested.

EXAMPLE 5:

```
      DIMENSION IRECPT{2},IREQBF{12},IRECBF{285}
C RETRIEVE RECORD FROM NEW ORDERS FILE
      IRECPT{1}=0
      IRECPT{2}=0
   10 IFLNUM=5
      CALL RTVSEQ{IFLNUM,0,IRECPT,IRECBF,285,IREQBF,IREQID}
C CHECK FOR ERRORS
      IF {IREQID.LT.0} GO TO 9000
C HAVE ALL RECORDS IN FILE BEEN READ
      IF{AND{IREQID,8}NE.0} GO TO 500
C WAS RECORD PREVIOUSLY REMOVED
      IF{AND{IREQID,$20}.NE.0} GO TO 10
C IS ROUTE DIFFERENT FROM THE DELIVERY ROUTE
C BEING LISTED
      IF {IRECBF{15}.NE.IROUTE} GO TO 10

C ROUTE MATCHES DELIVERY ROUTE BEING LISTED
C REMOVE RECORD FROM NEW ORDERS FILE
   20 CALL RTVDIR{IFLNUM,$8000,IRECPT,IRECBF,285,IREQBF,IREQID}
      IF {IREQID.LT.0} GO TO 9000

C PRINT INFORMATION ON DELIVERY ROUTE LIST
      .              .              .
      .              .              .
      .              .              .
C STORE RECORD IN ROUTED ORDERS FILE
      IFLNUM=17
      CALL STOSEQ{IFLNUM,IRECPT,IRECBF,285,IREQBF,IREQID}
C CHECK FOR ERRORS
      IF {IREQID.LT.0} GO TO 9000
C GO READ NEXT RECORD
      GO TO 10
  500 CONTINUE
```

3.3 INDEXED REQUESTS

3.3.1 STORE INDEXED RECORD

Records may be stored indexed in a file once it is defined as indexed. A record is stored in the file via its key value with its record pointer being returned to the caller so that the record may be accessed directly (refer to Section 3.4). An indexed store is permitted in a locked file with an indication made that the file was locked.

If a file was defined indexed-linked (refer to Section 3.1.2) more than one record may have the same key value. The record pointer of the last record to be stored (LIFO) or the next record to be stored (FIFO) with this key value is stored in the second and third words of the record. For indexed-linked files, each record must have these two words reserved exclusively for this use. Indexed-linked records must be at least three words long.

General information for store indexed records is as follows:

- The length of the record cannot exceed the specified maximum record length (refer to Section 2.6).
- If the file is not indexed-lined, not more than one record with the same key value can be stored.
- A record cannot be stored indexed if either the file is undefined or the file is not defined as indexed.
- An unprotected program cannot store a record in a file which was defined by a protected program.
- If a record is the first to be stored into an indexed-linked file with FIFO linking, the specified record length becomes the fixed record length for all subsequent records stored in the file. If the record is not the first and the file is indexed-linked with FIFO linking, the specified record length must be less than or equal to the length specified for the first record. (Even though the specified length of subsequent records may be less than the length of the first record, the fixed length will be used in storing all subsequent records.)

CAUTION

The store indexed request processor assumes that three words preceding and one word following the record buffer are part of the user's program.

The reason why the three words preceding the record buffer must be part of the user's program is explained in the cautionary note in Section 3.2.1. The word following the record buffer must be part of the user's program because as each record is stored into a file defined to have FIFO linking, the File Manager creates an extra record which reserves the space to be used for storing the next record. Thus, the pointer to the next record to be stored is immediately available for storage in words two and three. The extra record is stored with a header word containing $8000 which signifies that the record is an extra record created by the File Manager for a FIFO-linked file.

The value $8000 is temporarily stored into the word following the record buffer. After the record is transferred, the original contents of this word is restored. As for the three words preceding the buffer, the caution is necessary to prevent possible problems in a multiprogramming environment.

The store indexed record request specifies:

- The file number of the file where the record is being stored
- The key value of the record being stored
- A buffer for returning the record pointer
- A buffer for information to be stored as the record
- A temporary buffer for processing the request
- An indicator word denoting the request's status upon completion

The FORTRAN format for store indexed record call is as follows:

```
CALL STOIDX (filnum,keyval,recptr,recbuf,reclth,reqbuf,reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file into which a record is to be stored.
	keyval	is	the key value; it is an array of keylth words (refer to Section 3.1.2) containing the key value of the record.
	recptr	is	the record pointer; it is a two-word array set by the File Manager, which contains a record pointer to where the record was stored.
	recbuf	is	the record buffer; it is an array of reclth words containing the record to be stored.
	reclth	is	the record length. It contains a positive integer specifying the length of the record. If the record is the first to be stored into an indexed-linked file with FIFO linking, reclth becomes the fixed record length for all subsequent stores. If it is not the first store into the file, reclth must be less than or equal to the length of the first record. (Even though reclth may be less than the length of the first record, the fixed length will be used in storing all subsequent records).
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

STOIDX	filnum, keyval, recbuf	(Produces a call with an absolute address for each parameter)
STOIDX*	filnum, keyval, recbuf	(Produces a call with a relative address for each parameter, as for a run-anywhere program)

The parameters filnum, keyval, and recbuf are defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3.5.2.

3.3.2 RETRIEVE INDEXED RECORD

Records may be retrieved indexed from a file once the file is defined as indexed and at least one record has previously been stored indexed in the file. The file may have been defined by either a protected or an unprotected program. A record is retrieved from the file via its key value with its record pointer being returned to the caller so that the record may be accessed directly (refer to Section 3.4).

For update purposes, a record may be retrieved and the file locked with a file combination. More records may be retrieved from the locked file only if the same or no file combination is used. These retrieved and updated records may be restored in the locked file via the store direct record call, only if the same file combination is used. An indexed retrieve, which attempts to lock an already locked file with a different combination, is queued and not executed until the file becomes unlocked.

An indexed retrieve without a file combination is permitted from a locked file with an indication being given that the file was locked. Provision is also made for removing a record from a file as it is retrieved. The first part of a record of any desired length may be retrieved with an indication being given that there was a short retrieve.

If a record is retrieved and at least one more record with the same key value exists (which implies the file is indexed-linked), an indication is given that more records exist with the same key value. The continued execution of the RTVIDX call retrieves all the records with this key value. Following these retrievals, an end-of-link indication is returned in bit 4 of the parameter reqind to signify the end of the indexed-linked records with the same key value. Once such a repeated retrieve is initiated, new records which may be added to the link during this sequence of retrieves are ignored.

General information for retrieve indexed record is as follows:

- It is not necessary to retrieve all the index-linked records.
- The first and subsequent records with the same key value can be re-retrieved by a new call or by re-initializing the current call (refer to the recptr parameter).
- A record cannot be retrieved indexed if the record was previously removed from the file.
- A record cannot be retrieved indexed if either the file or the key was not defined.
- An unprotected program cannot remove a record from a file which was defined by a protected program.
- Records in an indexed-linked file must be a minimum of three words in length.

CAUTION

Parameters of a repeated RTVIDX call cannot be altered between calls unless a record with a different key is to be retrieved.

The retrieve indexed request specifies:

- The number of the file from where the record is being retrieved
- The key value of the record
- The file combination if the file is to be locked or the record is to be retrieved from a locked file
- Whether the record is to be removed from the file
- A buffer for returning the record pointer
- A buffer for receiving the record to be retrieved
- A temporary buffer for processing the request
- An indicator word denoting the request's status upon completion

The format for retrieve indexed record call is as follows:

```
CALL RTVIDX (filnum,keyval,filcom,recptr,recbuf,reclth,reqbuf,reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file from which a record is to be retrieved.
	keyval	is	the key value; it is an array of keylth words containing the key value of the desired record (see Section 3.1.2).
	filcom	is	the file combination with the remove option; bits 0 through 14 contain a non-zero number (if the file is or is to be locked) specifying the combination (which is or is to be) used to lock the file; bit 15 set to one indicates that the record is to be removed from the file.
	recptr	is	the record pointer; it is a two-word array set by the File Manager, which contains the record pointer from where the record was retrieved. If the file is indexed-linked, both words must initially be set to zero by the requestor.
	recbuf	is	the record buffer; it is a non-preset array of reclth words, where the File Manager transfers the retrieved record.
	reclth	is	the record buffer length; it contains a positive integer specifying the length of the record buffer.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

RTVIDX	filnum, keyval, recbuf	(Produces a call with an absolute address for each parameter)
RTVIDX*	filnum, keyval, recbuf	(Produces a call with a relative address for each parameter, as for a run-anywhere program)

The parameters filnum, keyval, and recbuf are defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3.5.2.

3.3.3 EXAMPLES OF INDEXED REQUESTS

A computer system in a medical clinic includes in its files a patient data file, indexed by social security number. The FORTRAN code in Example 6 shows how a record is stored indexed into the file. The patient's social security number is in words 1, 2, and 3 of the array ISOCSC. The key value is not part of the record data in this case.

EXAMPLE 6:

```
      DIMENSION IREQBF(12),IRECBF(31)
      DIMENSION IRECPT(2),ISOCSC(3)
       .      .
       .      .
       .      .
C PATIENT DATA HAS BEEN STORED IN IRECBF.
C STORE RECORD IN FILE.
      IFLNUM=15

      CALL STOIDX(IFLNUM,ISOCSC,IRECPT,IRECBF,31,IREQBF,IREQID)

C CHECK FOR ERRORS
      IF (IREQID.LT.0) GO TO 5000
```

A hospital laboratory computer system includes an indexed-linked file of laboratory test results. Each record in the file consists of the laboratory results for a given test for a given patient. The file is indexed by the patient's social security number. All the records for a given patient are FIFO-linked. The FORTRAN code in Example 7 is part of a program used to print a laboratory report on a given patient as requested by a physician. The retrieve indexed request is repeated until all records for the patient have been retrieved.

EXAMPLE 7:

```
      DIMENSION IREQBF{12},IRECBF{31},ISOCSC{3},IRECPT{2}
         .                          .
         .                          .
         .                          .
C SOCIAL SECURITY NUMBER IS IN ISOCSC ARRAY
C PROCEED TO PREPARE LAB REPORT ON PATIENT

      IFLNUM=25
C ZERO OUT RECORD POINTER ARRAY TO INDICATE
C THE START OF A LOOP TO RETRIEVE ALL RECORDS FOR THIS KEY VALUE
      IRECPT{1}=0
      IRECPT{2}=0

   10 CALL RTVIDX{IFLNUM,ISOCSC,0,IRECPT,IRECBF,31,IREQBF,IREQID}
C CHECK FOR ERRORS
      IF{IREQID.LT.0} GO TO 9000


C WAS RECORD PREVIOUSLY REMOVED-
      IF {AND{IREQID,$20}.NE.0} GO TO 10
C PROCESS DATA FROM THIS RECORD FOR REPORT
         .                          .
         .                          .
         .                          .
C GO BACK TO READ NEXT RECORD
C WAS THIS THE LAST RECORD FOR THIS PATIENT
  100 IF {AND{IREQID,$10}.EQ.0} GO TO 500
      GO TO 10

  500 CONTINUE
```

3.3.4 RETRIEVE INDEXED-ORDERED RECORD

Records may be retrieved indexed-ordered from a file once the file has been defined and it has been defined as indexed-ordered and at least one record has been previously stored indexed in the file. The file may have been defined by either a protected or an unprotected program. Each record in the file may be retrieved indexed-ordered by repeatedly executing one RTVIDO call until an end-of-file indication is given. If it is desired to retrieve records commencing with the record with a specific numeric key value, or if this record does not exist, the first record with a larger key value, the key value parameter has been set to a number less than the smallest key value, the first RTVIDO call will retrieve the record in the file that has the lowest numeric key value, the nth RTVIDO call will retrieve the record in the file that has the highest numeric key value, and the n + 1st RTVIDO call will produce an end-of-file indication. For each of the n calls, one record is retrieved, along with its key value and the corresponding record pointer so that the record may be accessed directly (see Section 3.4). If there are no records in the file, the first call will produce an end-of-file indication.

General information for retrieve indexed-ordered record is as follows:

- It is not necessary to retrieve all the records from the file.
- The record with the lowest key value or of a specified key value (and subsequent ordered records) can be re-retrieved by a new call or by re-initializing the current call (refer to the recptr parameter).
- For update purposes, a record may be retrieved and the file locked with a file combination. More records may be retrieved from the locked file only if the same or no file combination is used. These retrieved and updated records may be restored in the locked file via the store direct record call (refer to Section 3.4) only if the same file combination is used. An indexed-ordered retrieve, which attempts to lock an already locked file with a different combination, is queued and not executed until the file becomes unlocked.
- An indexed retrieve without a file combination is permitted from a locked file with an indication made that the file was locked. Provision is also made for removing a record from a file as it is retrieved. The first part of a record of any desired length may be retrieved with an indication made that there was a short retrieve.
- If a record is retrieved and at least one more record exists with the same key value (which implies the file is indexed-linked), an indication is made that more records exist with the same key value. The continued execution of the RTVIDO call retrieves all the records with this key value. Following these retrievals the File Manager proceeds as before.
- A record cannot be retrieved indexed-ordered if either the file or the key was not defined. Moreover, an unprotected program cannot remove a record from a file which was defined by a protected program.

CAUTION

Parameters of a repeated RTVIDO call cannot be altered between calls.

The retrieve indexed-ordered request specifies:

- The file number of the record from where the record is being retrieved
- A buffer for returning the key value
- The file combination, if the file is to be locked or the record is to be retrieved from a locked file
- Whether the record is to be removed from the file
- A buffer for returning the record pointer
- A buffer for receiving the record to be retrieved
- A temporary buffer for processing the request
- An indicator word denoting the request's status upon completion
- Records in an indexed-linked file must be a minimum of three words in length

The format for retrieve indexed-ordered record call is as follows:

```
CALL RTVIDO (filnum,keyval,filcom,recptr, recbuf, reclth, reqbuf, reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file from which a record is to be retrieved.
	keyval	is	the key value; it contains an integer equal to the lowest numeric key value desired, otherwise it contains a positive integer specifying the numeric key value of the desired record. Keyval will be set to the key value of the retrieved record by the File Manager.
	filcom	is	the file combination with the remove option; bits 0 through 14 contain a non-zero number (if the file is or is to be locked) specifying the combination (which is or is to be) used to lock the file; bit 15 set to one indicates that the record is to be removed from the file.
	recptr	is	the record pointer; it is a two-word array set by the File Manager, which will contain the record pointer from where the record was retrieved. Initially, both words must be set to zero by the requestor.
	recbuf	is	the record buffer; it is a non-preset array of reclth words, where the File Manager transfers the retrieved record.
	reclth	is	the record buffer length; it contains a positive integer specifying the length of of the record buffer.
	reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
	reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

RTVIDO	filnum, keyval, recbuf	(Produces a call with an absolute address for each parameter)
RTVIDO*	filnum, keyval, recbuf	(Produces a call with a relative address for each parameter, as for a run-anywhere program)

The parameters filnum, keyval, and recbuf are defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3.5.2.

3.3.5 EXAMPLE OF INDEXED-ORDERED REQUEST

The results of a psychological test have been coded and stored into an indexed-ordered file called the results file. Each record in the file contains test results for one subject. Each record is indexed by the age of the subject. All records for subjects with the same age are FIFO-linked. The age groups are ordered by the key value age. A schematic diagram of the contents of the file is shown in Figure 3-1.

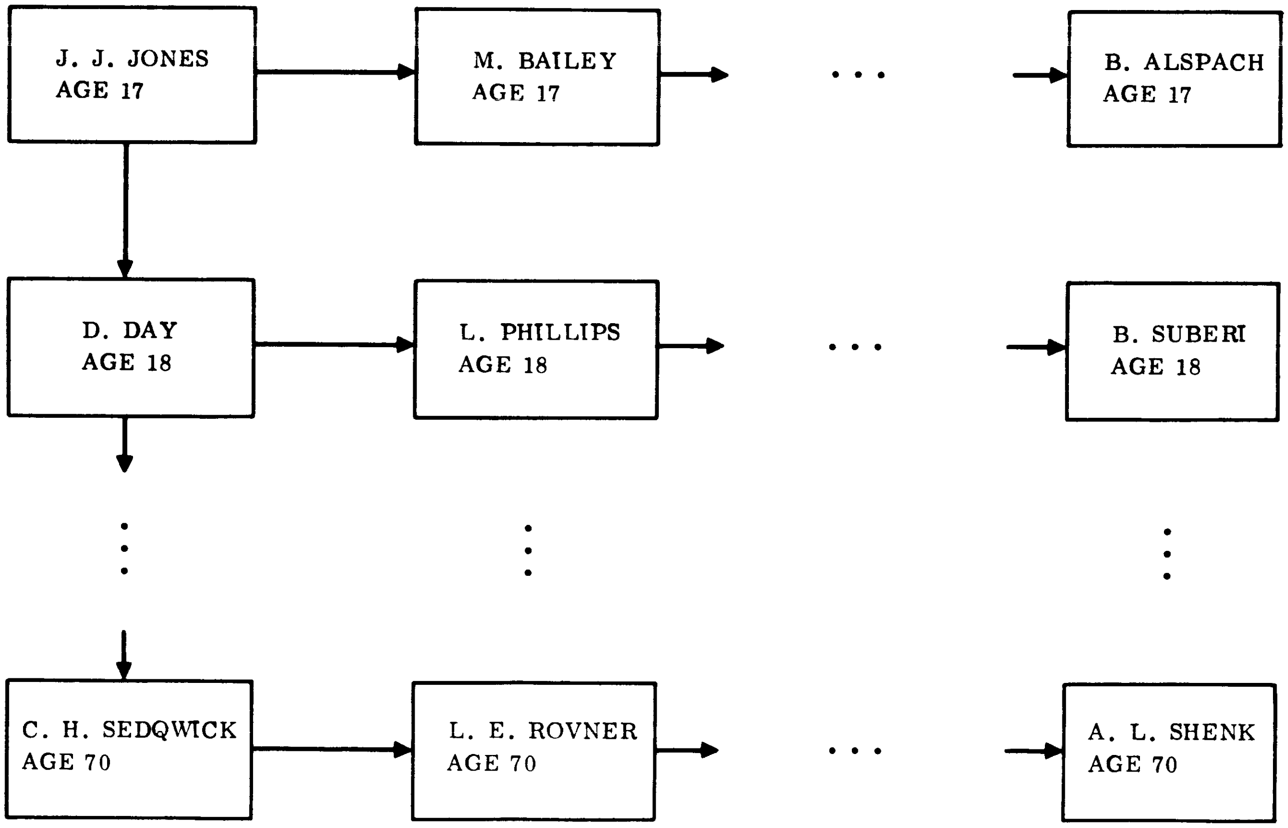


Figure 3-1. Schematic Representation of Indexed-Ordered, Indexed-Linked File Example

The FORTRAN code in Example 8 shows part of a program to do a statistical analysis on the results from all subjects with ages 25 through 35. The record pointer array is initially set to zero. Each set of calls for a given value of IAGE retrieves the results for all subjects with that age. If there are no subjects with that age, the first record for a larger age will be retrieved. To be sure that only records for subjects aged 25 to 35 are included in the data to be analyzed, a test is made on IAGE after a record is retrieved.

EXAMPLE 8:

```
      DIMENSION IRECPT(2),IREQBF(12),IRECBF(31)

C INITIALIZE TOTALS FOR STATISTICAL CALCULATIONS-
      ITOT1=0
      ITOT10=0

      IFLNUM=100
      IAGE=25
      IRECPT(1)=0
      IRECPT(2)=0
   10 CALL RTVIDO(IFLNUM,IAGE,0,IRECPT,IRECBF,31,IREQBF,IREQID)
C CHECK FOR ERRORS
      IF(IREQID.LT.0) GO TO 9000
C HAS RECORD BEEN REMOVED
      IF(AND(IREQID,$20).NE.0) GO TO 10
C CHECK THAT AGE IS WITHIN SPECIFIED RANGE
      IF (IAGE. GT. 35) GO TO 600
C PROCESS DATA FOR THIS RECORD
      ITOT1 =ITOT1 + IRECBF(12)
      ITOT2 =ITOT2 + IRECBF(13)
            .     .     .
            .     .     .
            .     .     .
      ITOT10 =ITOT10 + IRECBF(21)
C GO BACK TO READ NEXT RECORD
      GO TO 10
C PRINT  STATISTICAL RESULTS
  600 CONTINUE
```

3.4 DIRECT REQUESTS

3.4.1 STORE DIRECT RECORD

Records may be stored directly in a file once it is defined and the file has been locked by a retrieve request. A record is stored in the file through a record pointer previously provided when the record was either stored or retrieved by non-direct methods. The function of the store direct request is to update records.

General information for the store direct record is as follows:

- An update can be done only if the same file combination is supplied that was used to lock the file. This request also permits the file to be unlocked after the record has been updated and stored.
- A record cannot be stored directly if the file was not defined, not locked, or if the file combination is incorrect.
- An unprotected program cannot store direct because of the possibility of destroying a protected file by using an incorrect record pointer.

The store direct record request specifies·

- The number of the file where the record is being stored
- The file combination previously used to lock the file
- Whether the file should be unlocked
- A buffer containing the record pointer
- A buffer of information to be restored as the record
- A temporary buffer for processing the request
- An indicator word, denoting the request's status upon completion

The FORTRAN format for store direct record call is as follows:

```
CALL STODIR (filnum,filcom,recptr,recbuf,reqbuf,reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file in which a record is to be stored.
	filcom	is	the file combination and the file unlock option. Bits 0 through 14 contain a non-zero number which is identical to the combination that was used to lock the file; bit 15 set to one indicates that the file will be unlocked.
	recptr	is	the record pointer. It is a two-word array containing a pointer to where the record is to be stored.
	recbuf	is	the record buffer; it is an array of reclth words containing the record to be stored.

reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

STODIR	filnum, recbuf	(Produces a call with an absolute address for each parameter)
STODIR*	filnum, recbuf	(Produces a call with a relative address for each parameter, as for a run-anywhere program)

The parameters filnum and recbuf are defined as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program, as described in Section 3.5.2.

3.4.2 RETRIEVE DIRECT RECORD

The function of the retrieve direct request is to provide:

- A fast method of retrieving frequently accessed records
- A method for retrieving records linked together by their record pointers in the user's own list structure

Records may be retrieved directly from a file once the file is defined and at least one record has been stored in the file. The file may have been defined by either a protected or an unprotected program. A record is retrieved from the file through a record pointer previously provided when the record was either stored or retrieved by non-direct methods (refer to Sections 3.2 and 3.3).

General information for retrieve direct record is as follows.

- The user may form any number of complex list structures as long as they conform to the File Manager file structure. One list structure is provided for indexed-linked files.
- For LIFO-linked files, the record pointer of the last stored record with the same key value is stored by the File Manager in the second and third words of the record currently to be stored (see Section 3.3.1). These records may then be retrieved directly on a LIFO basis by referencing the second and third words of the last retrieved record as the record pointer. The end of the list is signified by the record pointer being zero (second and third words both zero). For the FIFO-linked files, the File Manager stores the record pointer of the next record with the same key value to be stored in the second and third words of the record currently to be stored. These records may then be retrieved directly on a FIFO basis by referencing the second and third words of the last retrieved record as the record pointer. The end of the list is signified when the request indicator for the retrieve direct request indicates that the record does not exist (see Section 3.4.2). This condition occurs because the File Manager sets the removed flag in the extra record (See Section 3.3.1).

- For update purposes, a record may be retrieved and the file locked with a file combination. More records may be retrieved from the locked file only if the same or no file combination is used. These retrieved and updated records may be stored into the locked file via the store direct record call, again only if the same file combination is used. Note that a direct retrieve, which attempts to lock an already locked file with a different combination, is queued and is not executed until the file becomes unlocked.
- A direct retrieve without a file combination is permitted from a locked file with an indication made that the file was locked. Provision is also made for removing a record from a non-indexed file as it is retrieved (caution should be exercised on removing records which are part of a list structure). The first part of a record of any desired length may be retrieved with an indication made that there was a short retrieve.
- A record may be retrieved but cannot be removed from an indexed file using a retrieve direct request. A record is not retrieved directly if the record was previously removed from the file. A record cannot be retrieved if the file is not defined. An unprotected program cannot remove a record from a file which was defined by a protected program.

The retrieve direct record request specifies:

- The number of the file from where the record is being retrieved
- The file combination, if the file is to be locked or the record is to be retrieved from a locked file
- Whether the record is to be removed from the file
- A buffer containing the record pointer
- A buffer for receiving the record to be retrieved
- A temporary buffer for processing the request
- An indication word, denoting the request's status upon completion

The FORTRAN format for retrieve direct record call is as follows:

```
CALL RTVDIR (filnum,filcom, recptr, recbuf, reclth, reqbuf, reqind)
```

Where:	filnum	is	the file number; it contains a positive integer identifying the file from which a record is to be retrieved.
	filcom	is	the file combination with the remove option; bits 0 through 14 contain a non-zero number (if the file is or is to be locked) specifying the combination (which is or is to be) used to lock the file; bit 15 set to one indicates that the record is to be removed from the file.
	recptr	is	the record pointer; it is a two-word array containing the record pointer pointing to the record to be retrieved.
	recbuf	is	the record buffer; it is a non-preset array of reclth words, where the File Manager transfers the retrieved record.
	reclth	is	the record buffer length; it contains a positive integer specifying the length of the record buffer.

reqbuf	is	the file request buffer; it is a non-preset array of 12 words which the File Manager uses to process the request.
reqind	is	the file request indicator word; the File Manager sets zero or more request indicator bits to one and clears the rest to zero (refer to the reqind parameter in Section 3.1).

The Assembly language macro format is as follows:

```
RTVDIR    filnum, recbuf    (Produces absolute code)

RTVDIR*   filnum, recbuf    (Produces a call for a run-anywhere program)
```

The parameters filnum and recbuf have the same definitions as in the FORTRAN call. Note that to use the macro call, an FLDF macro call must be included in the program as described in Section 3.5.2.

3.4.3 EXAMPLES OF DIRECT REQUESTS

In a hospital system a file called the patient file has been defined by a protected program. Patient records have been stored into the file.

The FORTRAN code in Example 9 is part of a protected program. In the program the file is searched for a given patient record. The file is then locked. This is necessary before the direct store request. The record is updated and stored back into the file via a direct store request. The file is also unlocked by the direct store request.

Note that if the code in Example 9 were part of an unprotected program, the patient file could not be updated, since the store direct request would then be illegal. Even if the patient file were defined as indexed, and if the key of the found record were known, a store indexed request could not be used to store the updated record since the original record would still be in the file. The original record could not be removed by a retrieve indexed request from an unprotected program.

EXAMPLE 9:

```
      INTEGER FILNUM,FILCOM,RECPTR,RECBUF,RECLTH,REQBUF,REQIND,PATLOC
      DIMENSION RECPTR{2},RECBUF{41},REQBUF{12}
C RETRIEVE DESIRED RECORD FROM FILE 10
      FILNUM = 10
C SET THE FILE COMBINATION TO ZERO {NO FILE COMBINATION}

      FILCOM = 0
C CLEAR RECORD POINTER TO INITIALIZE THE RETRIEVE REQUEST
      RECPTR{1} = 0
      RECPTR{2} = 0
```

EXAMPLE 9 (Continued):

```
C USER TO RETRIEVE A RECORD 41 WORDS LONG
      RECLTH = 41
C RETRIEVE RECORDS SEQUENTIALLY FROM FILE 10 LOOKING FOR
C PATIENT RECORD
 1000 CALL RTVSEQ{FILNUM,FILCOM,RECPTR,RECBUF,RECLTH,
      REQBUF,REQIND}
C GO TO 9998 IF SEQUENTIAL RETRIEVE ERROR
      IF {REQIND.NE.0} GO TO 9998
C
C
C
C THE RECORD FORMAT IS THE RECORD LENGTH IN WORD 1, TWO TWO-
C WORD RECORD POINTERS IN WORDS 2 THRU 5, THE PATIENT LOCAT-
C ION IN WORD 6,  THE MEDICATION CODE IN WORD 7, AND PATIENT
C DATA IN WORDS 8 THRU 41
C
C IF PATIENT LOCATION DOES NOT MATCH.  CHECK NEXT RECORD
      IF {RECBUF{6}.NE.PATLOC} GO TO 1000
C
C
C
C PATIENT RECORD WITH THE GIVEN PATIENT LOCATION FOUND
C GENERATE UNIQUE FILE COMBINATION TO LOCK FILE
C NOTE**ONLY THIS METHOD SHOULD BE USED TO GENERATE FILE COMBINATIONS**
 2000 ASSIGN 2000 TO FILCOM
C
C
C
C LOCK THE FILE FOR UPDATE
      CALL LOKFIL{FILNUM,FILCOM,REQBUF,REQIND}
C GO TO 9997 IF LOKFIL ERROR
      IF {REQIND.NE.0} GO TO 9997
C
C
C
C UPDATE PATIENT RECORD WITH MEDICATION CODE
      RECBUF {7} = MEDCOD
C SET FILE COMBINATION TO UNLOCK THE FILE AFTER THE UPDATE
      FILCOM = FILCOM + $8000
C STORE UPDATED RECORD FOR PATIENT DIRECTLY BACK INTO FILE 10
      CALL STODIR{FILNUM,FILCOM,RECPTR,RECBUF,REQBUF,REQIND}
C GO TO 9996 IF DIRECT STORE ERROR
      IF {REQIND.NE.0} GO TO 9996
C
C
C
```

In Example 5 we had an example of a direct retrieve request used to remove a record from a non-indexed file. Note in this example that if file 5 had been an indexed file, the direct retrieve request could not have been used to remove the record. If file 5 were indexed and if the key value were contained in the record data, the key value could be extracted from the record and used to retrieve and remove the desired record by means of a retrieve indexed request.

Statement 20 in the code listed in Example 5 would be changed in the case of an indexed file. The corresponding FORTRAN code for an indexed file is shown in Example 10.

<u>EXAMPLE 10</u>:

```
C REMOVE RECORD FROM NEW ORDERS FILE.
C KEY VALUE IS CONTAINED IN
C WORDS 41 AND 42 OF RECORD
   20 CALL RTVIDX(IFLNUM,IRECBF(41),$8000,IRECPT,IRECBF,
     285,IREQBF,IREQID)
C CHECK FOR ERRORS
      IF(IREQID.LT.0) GO TO 9000
```

3.5 ASSEMBLY LANGUAGE COMMUNICATION WITH THE FILE MANAGER

3.5.1 CALLING SEQUENCES WITHOUT USE OF MACROS

Calling sequences written in Assembly language which are intended to communicate with File Manager subprograms may have the following form, where flrqst is the name of a File Manager routine, and flrqst is declared as an external in the user's program.

LOC	RTJ flrqst
LOC+1	(RTJ flrqst is a two-word instruction)
LOC+2	Address of argument 1
LOC+3	Address of argument 2
LOC+4	Address of argument 3
•	
•	
•	
LOC+N	Address of argument N
LOC+N+1	Program resumes

3.5.2 USE OF FLDF MACROS

Macros from the macro library, as described in Sections 3.1, 3.2, 3.3, and 3.4, may be used to generate File Manager calls. When one or more macro is used in a program to make a File Manager call, an FLDF macro call must be included in the program unit. The FLDF macro does not generate executable code; therefore the call to FLDF must be positioned in the program so that it will not be executed. For example, it could be placed in a subprogram before the first entry point, at the end of a program, or within the body of the program preceded by a jump instruction to bypass the nonexecutable code in the macro. The format of the FLDF macro call is as follows:

```
FLDF     filnum, maxrl, lu, numekv, keylth, filcom, reclth
```

The parameters are defined as in Sections 3.1, 3.2, 3.3, and 3.4. Each parameter in the calling list may be a constant or a variable. If it is a variable and its value is required by a File Manager call, the specific value must be included in the macro call or stored into the variable location prior to the corresponding macro call. If the lu position in the calling list is blank, the logical unit is set to 8, the normal library unit. An example of an FLDF call follows:

```
FLDF     FILNUM, MAXRL,,250, 1, 0, RECLTH
```

3.5.3 A MACRO TO TEST REQUEST INDICATOR BITS ON RETURN FROM A FILE MANAGER CALL

The file status macro, STATFL, provides the user with an easy method of getting the request indicator word, masking specified error conditions, and giving control to a specified error routine if errors are present. A file status of zero implies that no errors occurred on the last file request. Forms of the macro call are as follows:

```
STATFL      fn, mk, bd
STATFL      fn
STATFL      fn, mk
STATFL      fn, mk, bd
```

Where:	fn	is	the file number
	mk	is	the mask that is used to form the logical product with the request indicator. (If mk is left blank, only the status is placed in the A register.) The terminator, such as the dash (-) in the fourth example, determines the addressing mode used on the AND instruction and may be a -, +, *, or blank.
	bd	is	the program label where control is given if the logical product of mk and the request indicator is non-zero. If bd is left blank, no code will be generated to test the request indicator status. In this case the logical product of the request indicator and the mask is left in the A register at the end of the macro and may be tested by the user.

TIME REQUIREMENTS 4

In this section, the access rate equations are given in terms of the number of accesses for each storage/retrieval method and the transfer rate for the mass memory device. Next, an example illustrates the calculation of the access rates. The equations for the number of accesses and transfer rates are derived in Appendix C from the definition of the file structure in Appendix A and the characteristics of the mass memory devices (disk and drum).

DEFINITIONS

The following terms will be used in this section:

- Seek time — The time required for the disk arm to travel from a given disk cylinder to the disk cylinder to be accessed. (A drum has no seek time.)
- Latency — The time required for a mass memory device to travel from the data to be accessed to the read heads within a given track.

To the definitions given in Sections B.1.1 and B.3.1, the following symbol definitions are added:

- AR — Access rate
- NA — Number of accesses
- TR — Transfer rate
- SS — For sequential store
- NSR — For next sequential retrieve
- ASR — For any sequential retrieve
- IS — For indexed store
- IR — For indexed retrieve
- DS — For direct store
- DR — For direct retrieve
- DK — For disk
- DKR — For disk read
- DKW — For disk write
- DM — For drum
- f(ro) — Remove option function (a one if the record is to be removed, otherwise zero)

- $\overline{\text{SEEK}}$ — Average seek time of the disk
- $\overline{\text{LAT}}$ — Average latency of mass memory device
- LAT — Maximum latency of mass memory device
- TWR — Transfer word rate of mass-memory device
- IUNF — Initial usage for non-indexed file
- IUIF — Initial usage for indexed file
- TUNF — Terminal usage for non-indexed file
- TUIF — Terminal usage for indexed file
- NFISSS — Number of file information segments with the same scatter code

ASSUMPTIONS

The access rate equations are approximations in that they assume the File Manager software is in core and its overhead is negligible, and that the initial and terminal usage (see Sections C.1.4 and C.1.5) of a file can be neglected if a fairly large number of records are accessed. Note that it is also assumed that the record lengths are small (e.g., $RL \leq 93$) for the disk transfer rates, that there are no key information segment overflow blocks (to simplify the derivation), and that the assumptions of Section B.1.2 are true.

4.1 ACCESS RATE EQUATIONS

The access rate for any of the storage/retrieval methods is found by taking the product of the number of accesses required by that method and the transfer rate of the mass memory device, i.e.,

$$AC = NA \cdot TR$$

4.1.1 SUMMARY OF ACCESS EQUATIONS

The equations in this section are derived in Appendix C. Note that some of the access rate equations in this section give access rates for record storage. These equations assume that any needed space will be available in the file space list. If this is not the case, additional accesses from the file space pool will be necessary to obtain the space. These additional accesses may significantly add to the number of accesses. To maximize use of the file space list and thereby minimize the number of file space pool accesses, refer to Section 2.7.1.

The following summary is based on derivations from Section C.1 and assumptions from the beginning of this section.

1. Number of accesses to store a sequential record as a part of a loop to store all the sequential records in one file record block:

$$NA_{SS} = 1 + \frac{RL}{MAXRL}$$

2. Number of accesses to retrieve the next sequential record:

$$NA_{NSR} = 1 + f(ro)$$

3. Average number of accesses to retrieve any sequential record:

$$NA_{ASR} = \frac{NR}{2} + f(ro)$$

4. Number of accesses to store an indexed record as a part of a loop to store all the indexed records in a file record block:

$$NA_{IS} = 3 + \frac{RL}{MAXRL}$$

5. Number of accesses to retrieve an indexed record:

$$NA_{IR} = 2 + 2 \cdot f(ro)$$

6. Number of accesses to store a direct record:

$$NA_{DS} = 1$$

7. Number of accesses to retrieve a direct record:

$$NA_{DR} = 1 + f(ro)$$

4.1.2 SUMMARY OF DISK/DRUM TRANSFER RATES

The following derivations are from Section C.2 with assumptions from the beginning of this section.

1. Average transfer rate for disk read:

$$TR_{DKR} \approx 123 + \frac{25}{96}(RL - 1) \text{ milliseconds/access}$$

2. Average transfer rate for disk write:

$$TR_{DKW} \approx 148 + \frac{50}{96}(RL - 1) \text{ milliseconds/access}$$

3. Average transfer rate for drum:

$$TR_{DM} = 8 + .008 \cdot RL \text{ milliseconds/access}$$

4.1.3 ACCESS RATE EQUATIONS FOR DISK

From Sections 4.1.1 and 4.1.2:

1. Access rate of sequential store as a part of a loop to sequentially store records in the file:

$$AR_{SSDK} \approx \left(1 + \frac{RL}{MAXRL}\right)\left(148 + \frac{50}{96}(RL - 1)\right)$$

2. Access rate of next sequential retrieve:

$$AR_{NSRDK} \approx \left(1 + f(ro)\right)\left(123 + \frac{25}{96}(RL - 1)\right)$$

3. Access rate of any sequential retrieve:

$$AR_{ASRDK} \approx \left(\frac{NR}{2} + f(ro)\right)\left(123 + \frac{25}{96}(RL - 1)\right)$$

NOTE

In equations 4 and 5 it is assumed that the length of any KIS block does not exceed 96 words. The time to access one word on disk is 12.8 microseconds. The time to access one word on drum is 8 microseconds. If the KIS block size is large, the total access rate may be significantly increased.

Consider the extreme case of a KIS block size equal to 12,000 words: Refer to Section B.4.3.5. The time needed to access all words in the KIS block would be 154 milliseconds if using a disk, or 96 milliseconds if using a drum.

4. Access rate of indexed store as a part of loop of indexed stores:

$$AR_{ISDK} \approx \left(3 + \frac{RL}{MAXRL}\right)\left(148 + \frac{50}{96}(RL - 1)\right)$$

5. Access rate of indexed retrieve as a part of a loop of indexed retrieves:

$$AR_{IRDK} \approx \left(2 + 2 \cdot f(ro)\right)\left(123 + \frac{25}{96}(RL - 1)\right)$$

6. Access rate of direct store:

$$AR_{DSDK} \approx 148 + \frac{50}{96}(RL - 1)$$

7. Access rate of direct retrieve:

$$AR_{DRDK} \approx \left((1 = f(ro)\right)\left(123 + \frac{25}{96}(RL - 1)\right)$$

4.1.4 ACCESS RATE EQUATIONS FOR DRUM

From Sections 4.1.1 and 4.1.2:

1. Access rate of sequential store as a part of a loop of sequential stores:

$$AR_{SSDM} = \left(1 + \frac{RL}{MAXRL}\right)\left(8 + .008 \cdot RL\right)$$

2. Access rate of next sequential retrieve:

$$AR_{NSRDM} = (1 + f(ro))\ (8 + .008 \cdot RL)$$

3. Access rate of any sequential retrieve:

$$AR_{ASRDM} = \left(\frac{NR}{2} + f(ro)\right)\ (8 + .008 \cdot RL)$$

4. Access rate of indexed store as a part of a loop to store indexed records:

$$AR_{ISDM} = \left(3 + \frac{RL}{MAXRL}\right)(8 + .008 \cdot RL)$$

5. Access rate of indexed retrieve:

$$AR_{IRDM} = (2 + 2 \cdot f(ro))\ (8 + .008 \cdot RL)$$

6. Access rate of direct store:

$$AR_{DSDM} = 8 + .008 \cdot RL$$

7. Access rate of direct retrieve:

$$AR_{DRDM} = (1 + f(ro))\ (8 + .008 \cdot RL)$$

4.2 EXAMPLE OF ACCESS RATE CALCULATIONS

One hundred records are retrieved, indexed, updated, and stored direct on disk. What is the time required if the record length is 19 words?

From Section 4.1.3, equation 5:

$$AR_{IRDK} \approx (2 + 2 \cdot f(ro)) \left(123 + \frac{25}{96} (RL - 1)\right)$$

$$\approx (2 + 0) \left(123 + \frac{25}{96} (19 - 1)\right)$$

$$\approx 256 \text{ MS/IS}$$

From Section 4.1.3, equation 6:

$$AR_{DSDK} \approx 148 + \frac{50}{96} (RL - 1)$$

$$\approx 148 + \frac{50}{96} (19 - 1)$$

$$\approx 157 \text{ MS/DS}$$

Therefore, the total time (T) required for this example is:

$$T \approx 100 (256 + 157)$$

$$T \approx 41.3 \text{ seconds}$$

4.3 MINIMIZATION OF TIME REQUIRED FOR INITIAL FILE ACCESS

The access times in Sections 4.1.3, 4.1.4, and 4.2 do not include accesses due to the initial usage of a file as described in Section C.1.4. These may be significant if a large number of different files are accessed. To minimize the number of initial accesses, the following procedure may be incorporated into system initialization.

The search to obtain a file's FIS can be minimized if the files are first defined in a particular sequence. This can be done by placing the FISs of the files with the same hash code in the same FIS block; there are 47 FIS pointers in the FIS directory and 17 FISs in the FIS block. Assuming a system has less than 800 (47 x 17 = 799) files, all the FISs with the same hash code can be placed in the same FIS block by defining the files in the following order:

1.	Files 1, 48, 95, . . . , 753	(17 defines)
2.	Files 2, 49, 96, . . . , 754	"
3.	Files 3, 50, 97, . . . , 755	"
.		.
.		.
.		.
47.	Files 47, 94, 141, . . . , 799	(17 defines)

If there are more than 800 files, a similar procedure can be developed.

If a user wishes to define the files dynamically, an initialization program can define them as explained above and then release them. Since the space for FIS blocks is not re-used, the order will then be determined and the number of accesses (two) to retrieve any FIS will be minimized.

FILE STRUCTURE A

Each defined file has a file information segment (FIS) that points to file record blocks (FRBs). An FRB contains file records and can be searched sequentially to access desired record(s). Alternately, a key information segment (KIS) can be used via the FIS to access one record randomly without searching through all the records. This appendix discusses each of these structures in detail.

A.1 FILE INFORMATION SEGMENT STRUCTURE

The file information segment (FIS) structure, located on mass memory, is composed of one FIS directory and zero or more FIS blocks (see Figure A-1). Information to/from this structure is obtained via five core-resident parameters:

• FIDSEC	is	the FIS directory's sector address.
• NWFISD	is	the number of words in the FIS directory (multiple of 96).
• FIBLSA	is	the sector address of the last FIS block.
• FIBNIX	is	the index to the next available location in FIBLSA.
• NWFISB	is	the number of words in a FIS block (multiple of 96).

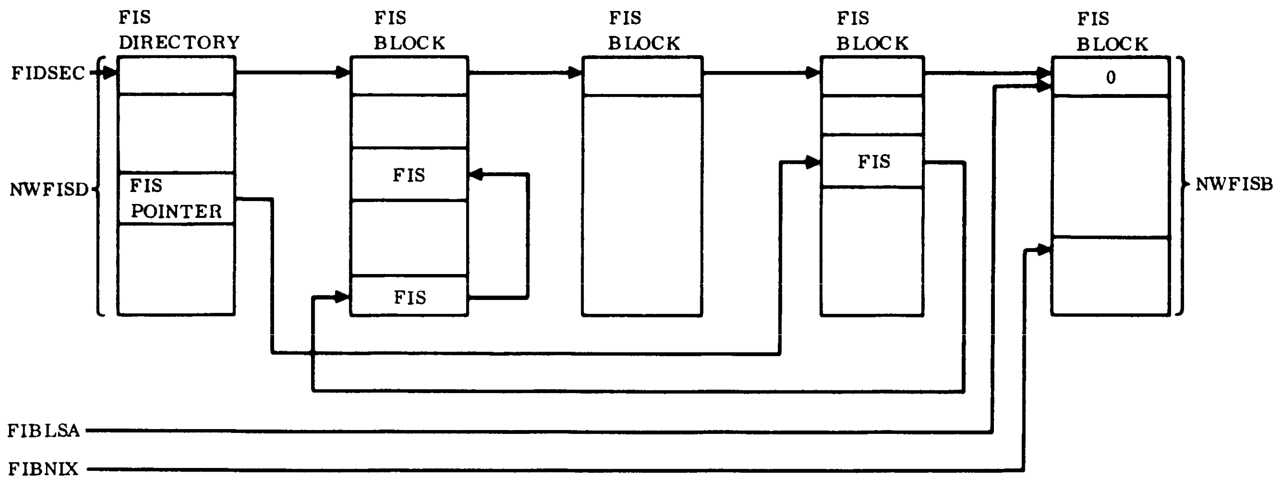


Figure A-1. File Information Segment Structure

A.1.1 FILE INFORMATION SEGMENT DIRECTORY

The FIS directory of NWFISD words is created on mass memory when the first file is defined. Its sector address is given by the core-resident parameter FIDSEC. The directory is composed of:

- A two-word header, which contains the sector address of the first FIS block (a zero indicates there are no FIS blocks) and a word reserved for future use
- Up to $\lceil(\text{NWFISD} - 2)/2\rceil$ two-word FIS pointers

Utilizing the file number, modulo $\lfloor(\text{NWFISD}-2)/2\rfloor$, for a scatter code, each pointer points to the first FIS with a scatter code corresponding to the pointer's relative position in the FIS directory. If a FIS pointer (both words) is zero, there are no FISes for that particular scatter code. The FIS pointer has the same format as a record pointer (see Section 2.3).

A.1.2 FILE INFORMATION SEGMENT BLOCK

A FIS block of NWFISB words is created on mass memory whenever space is needed to store a newly defined FIS. Its sector address is given either by the FIS directory (if it is the first FIS block) or by a previously allocated FIS block. The block is composed of:

- A header word, containing the sector address of the next FIS block (a zero indicates there are no more FIS blocks)
- Up to $\lfloor(\text{NWFISB} - 1)/16\rfloor$ FISs (see Section A.1.3)

There are also two core-resident parameters, FIBLSA and FIBNIX, which give the sector address of the last FIS block and the index to the next available location in the last FIS block respectively.

A.1.3 FILE INFORMATION SEGMENT

A 16-word FIS is stored into a FIS block whenever a file is defined; its two-word mass memory address is given by a FIS pointer in either the FIS directory (if it is the first FIS with a particular scatter code) or a previously stored FIS. Note that once a file is defined, its FIS exists permanently, even if the file is released. A FIS is composed of the following:

Word	Mnemonic	Description
0	SANFIS	Sector address of next FIS with the same scatter code (if zero, there are no more FISs for this particular scatter code)
1	IXNFIS	Index into SANFIS to next FIS with the same scatter code

†Where $\lfloor x\rfloor$ is the greatest integer less than or equal to x. (see Section B.4.2).

Word	Mnemonic	Description
2	FILENO	File number
3	FRBFSA	Sector address of the first file record block (a zero indicates there are no file record blocks)
4	NRLFRB	Number of records stored in the last FRB
5	FRBLSA	Sector address of the last file record block
6	FRBNIX	Index to the next available location in FRBLSA
7	KIDSEC	Key information segment (KIS) directory's sector address (a zero indicates there is none)
8	KIDSIZ	KIS directory's size in sectors
9	KIBSIZ	KIS block size in sectors (a zero indicates the file is not indexed)
10	KEYLTH	Key length in words (a zero indicates the file is not indexed)
11	NUMEKV	Number of expected key values (a zero indicates the file is not indexed)
12	FIFORL	Fixed record length for indexed-linked FIFO file (a zero indicates that the file is not indexed-linked FIFO)
13	NUMFRB	Number of file record blocks currently assigned to the file
14	FRBSIZ	File record block size in sectors (bits 0 through 8)
	FISIND	FIS indicator with the following definition: Bit 13 is 0 File is indexed-linked LIFO. 1 File is indexed-linked FIFO. Bit 14 is 0 File is not indexed-ordered. 1 File is indexed-ordered. Bit 15 is 0 File is not indexed-linked. 1 File is indexed-linked.
15	FISFLG	FIS flag with the following definition: Bits 0 — 6 Logical unit for allocating FRBs Bits 7 — 13 Logical unit for allocating the KIS directory and KIS blocks Bit 14 is 0 Defined by an unprotected program 1 Defined by a protected program Bit 15 is 0 File is released. 1 File is defined.

A six-word header is appended to a FIS when the FIS is in core. A core FIS header is composed of the following:

Word	Mnemonic	Description
0	ANCFIS	Address of next core-resident FIS (a zero indicates this is the last)
1	SECFIS	Sector address of the FIS
2	IDXCHC	Index and change flags with the following definition: Bit 0 is 0 — FIS has not been changed. 1 — FIS has been changed. Bit 1 is 0 — KIS directory has not been changed. 1 — KIS directory has been changed. Bits 7 — 15 — Index to start of FIS from start of sector
3	ADRKID	Core address of KIS directory
4	FILCOM	File combination (zero if file not locked)
5	FILCLK	File clock (used for releasing FIS after a period of no activity)

A.2 FILE RECORD BLOCK STRUCTURE

The file record block (FRB) structure, located on mass memory, is composed of zero or more FRBs for each file (Figure A-2). Information to/from this structure is obtained via five parameters of the file's FIS (see Section A.1.3, words 3, 4, 5, 6, and 14):

- FRBFSA is the sector address of the first FRB.
- NRLFRB is the number of records stored in the last FRB.
- FRBLSA is the sector address of the last FRB.
- FRBNIX is the index to the next available location in FRBLSA.
- FRBSIZ is the file record block size.

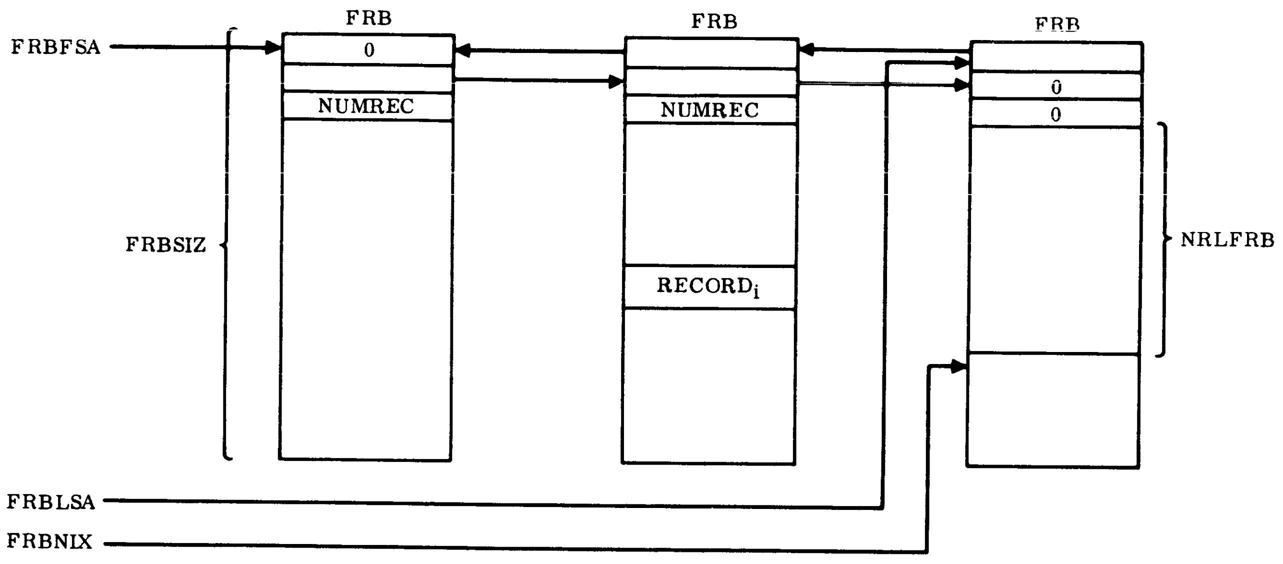


Figure A-2. File Record Block Structure

A.2.1 FILE RECORD BLOCK

An FRB of FRBSIZ sectors is created on mass memory whenever space is needed to store a new record. Its sector address is given either by the FIS (if it is the first FRB in a file) or by a previously allocated FRB. The size of an FRB, FRBSIZ, is specified by the computation of:

$$\left\lceil \frac{3 + \text{MAXRL}}{96} \right\rceil^{\dagger}$$

Where: MAXRL is the maximum record length of any record to be stored in the file.

The block is composed of:

- A three-word header containing:

  -The sector address of the last FRB (a zero indicates the first FRB)

  -The sector address of the next FRB (a zero indicates there are no more FRBs for this file)

  -The number of records stored in this FRB (a zero indicates that this is the last FRB and reference should be made to NRLFRB)

- Zero or more variable or fixed length records

†Where $\lceil x \rceil$ is the least integer greater than or equal to x.

There are also three other FIS parameters, NRLFRB, FRBLSA, and FRBNIX, which give the number of records stored in the last FRB, the sector address of the last FRB and the index to the next available location in the last FRB respectively, of each file.

A.2.2 FILE RECORD

A file record, or simply a record, of variable or fixed length is stored/retrieved into an FRB whenever a legal file request is given. Its length, variable or fixed, depends on the type of file: not indexed-linked or indexed-linked with or without FIFO linking. Its access depends on the type of file: indexed or not indexed. In general, a record is composed of:

- A header word containing:

  -The total length of the record in bits 0 through 14

  -The removed flag in bit 15 (the record has been removed from the file if bit 15 is one)
- A two-word record pointer if the file is indexed-linked
- Zero or more data words

The total record length is given by:

RL = 1 + NRPW + NDW

Where:	RL	is	the total record length.
	NRPW	is	the number of record pointer words (zero if not indexed-linked, two if indexed-linked).
	NDW	is	the number of data words.

If the record is an extra record created by the File Manager to reserve space for storage of a subsequent FIFO-linked record, the header word will contain $8000 and no useful information will have been stored in the remainder of the record. The length of this record will be specified by FIFORL in the associated file's FIS.

A.3 KEY INFORMATION SEGMENT STRUCTURE

The key information segment (KIS) structure, located on mass memory, is composed of:

- One KIS directory
- Zero or more KIS blocks for each file with a key defined area (see Figure A-3)

Information to/from this structure is obtained via five parameters of the file's FIS (words 7 through 11):

- KIDSEC is the KIS directory's sector address.
- KIDSIZ is the KIS directory's size in sectors.
- KIBSIZ is the KIS block's size in sectors.
- KEYLTH is the key length.
- NUMEKV is the number of expected key values.

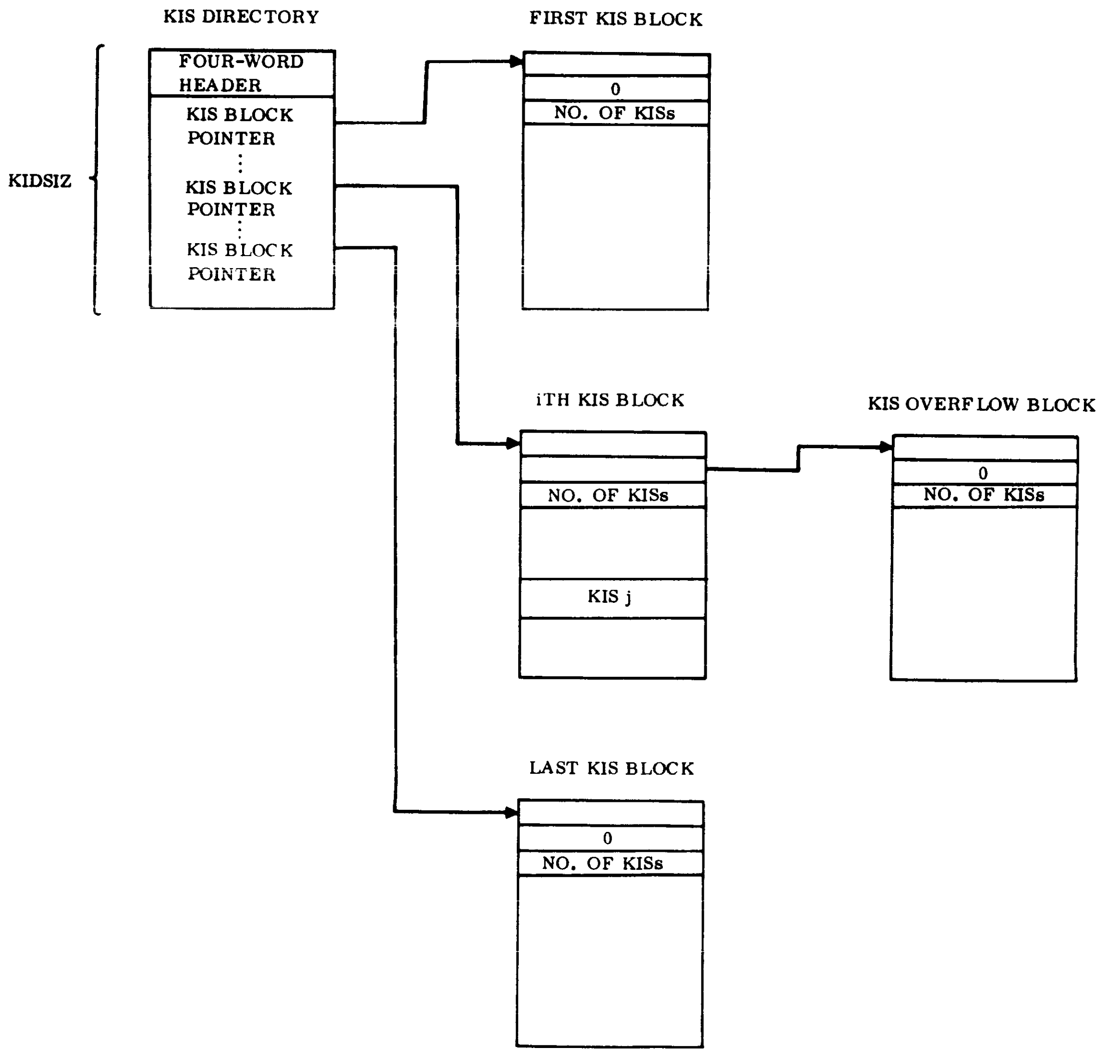


Figure A-3. Key Information Segment Structure

A.3.1 KEY INFORMATION SEGMENT DIRECTORY

The KIS directory of KIDSIZ sectors is created on mass memory whenever a file is defined to have a key (i.e., indexed). Its sector address is given by the parameter KIDSEC. The size of the KIS directory, KIDSIZ, is specified by the computation of

$$\lceil (4 + \text{SRNEKV})/96) \rceil$$

Where: SRNEKV is the square root of the number of expected key values (see Section 3.1.2).

The directory is composed of a four-word header, containing:

Word	Mnemonic	Description
0	KIDCLK	KIS directory clock
1	NUMKIB	Number of KIS blocks
2	KIBFSA	First sector address of linked KIS blocks
3	KIBLSA	Last sector address of linked KIS blocks

and up to KIBSIZ · 96 - 4 KIS block pointers. Utilizing the given key value to produce a scatter code, each KIS block pointer contains a one-word sector address of a KIS block with a scatter code corresponding to the pointer's relative position in the KIS directory. If a KIS block pointer is zero, there is no KIS block for that particular scatter code.

A.3.2 KEY INFORMATION SEGMENT BLOCK

A KIS block of KIBSIZ is created on mass memory whenever space is needed to store a file's record with a new key value. Its sector address is given by the corresponding KIS block pointer in the KIS directory of the file. The size of a KIS block, KIBSIZ, is specified by the computation of

$$\left\lceil \frac{3 + (2 \cdot \text{NUMPTR} + \text{KEYLTH}) \cdot \text{SRNEKV}}{96} \right\rceil$$

Where:	NUMPTR	is	the number of record pointers in each KIS. The value of NUMPTR is one for a LIFO linked or unlinked file. The value is two for a FIFO linked file.
	KEYLTH	is	the key length (in words).
	SRNEKV	is	the square root of the number of expected key values.

The block is composed of:

- A three-word header containing:

  -The sector address of the KIS block allocated before this one (a zero indicates the first KIS block)

  -The sector address of the first KIS overflow block (a zero indicates there are no KIS overflow blocks)

  -The number of KISs in the KIS block

- Up to $\lfloor(\text{KIBSIZ} \cdot 96 - 3)/(2 \cdot \text{NUMPTR} + \text{KEYLTH})\rfloor$ KISs

NOTE

An indexed-ordered file will cause each KIS block to be ordered by key value. The ith KIS block will contain key values in the range:

$$\text{NUMEKV}\left(\frac{i-1}{\text{NKISB}}\right) < \text{KEYVAL} \leq \text{NUMEKV}\left(\frac{i}{\text{NKISB}}\right)$$

Where:	NUMEKV	is the number of expected key values.
	NKISB	is the number of KIS blocks.
	KEYVAL	is the key value.

A.3.2.1 KEY INFORMATION SEGMENT OVERFLOW BLOCK

A KIS overflow block, KIBSIZ, is created on mass memory whenever a KIS block becomes filled, either because the number of expected key values was exceeded or the key values do not scatter uniformly. Its sector address is given by either its KIS block or by a previously allocated KIS overflow block. The block has the same format as a KIS block and is composed of:

- A three-word heading containing:

  -The sector address of the KIS block allocated before this one.

  -The sector address of the next KIS overflow block (a zero indicates there are no more KIS overflow blocks)

  -The number of KISs in the KIS overflow block

- Up to $\lfloor(\text{KIBSIZ} \cdot 96 - 3)/(2 \cdot \text{NUMPTR} + \text{KEYLTH})\rfloor$ KISs

In an indexed-ordered file, which is not indexed-linked, KIS blocks other than the first and last will not have overflow blocks. The first KIS block will have associated overflow blocks only if one or more records with negative key values are stored. The last KIS block will have associated overflow blocks only if one or more records with key values exceeding NUMEKV are stored. If overflow blocks are created for an indexed-ordered file, the order of the KISs with respect to key value is maintained in the KIS blocks and the KIS overflow blocks.

A.3.3 KEY INFORMATION SEGMENT

A KIS of 2 · NUMPTR + KEYLTH words is stored into a KIS block whenever a legal indexed file request is given to store a record with a new key value. Its access is achieved by searching the proper KIS block. A KIS for a LIFO-linked file record is composed of:

- A record pointer that points to the last record stored using the same key value
- A KEYLTH word array containing the key value

A KIS for a FIFO-linked file record is composed of:

- A record pointer that points to the first record stored using the same key value
- A record pointer that points to the last record stored using the same key value
- A KEYLTH word array containing the key value

A.3.4 ALLOCATION OF SPACE WITHIN THE KEY INFORMATION STRUCTURE

The File Manager is designed so that the size of the KIS directory is dependent on the number of expected key values, NUMEKV. We have seen that

$$\text{NWKISD} = \left\lceil \frac{\text{SRNEKV} + 4}{96} \right\rceil \cdot 96$$

For values of NUMEKV less than or equal to 92^2(=8464), 96 words (one sector) are used as the KIS directory. For NUMEKV values between 8,465 and 32,767, the KIS directory has a length of 192 words or two sectors. The hash code technique for scattering the key values into the KIS blocks is as follows:

Let

$$\{\text{KEYVAL}_{(i)};\ i = 1,\ 2,\ 3,\ \ldots\ \text{KEYLTH}\}$$

be the set of words comprising the key values for a given record. Let NEKISD equal the number of entries in the KIS directory (either 92 or 188), then H, the hash code for the record, is computed as

$$H = \left| \sum_{i=1}^{\text{KEYLTH}} \text{KEYVAL}_{(i)} \right| \pmod{\text{NEKISD}}$$

EXAMPLE:

Suppose a file is indexed with a key of location code (let KEYLTH = 2 and NUMEKV = 10,000) and a record is to be stored with the location code LJ11 ($ = a hexadecimal number), then the key value in ASCII is:

$$\text{KEYVAL}(1) = 4C4A_{16}$$

$$\text{KEYVAL}(2) = 3131_{16}$$

then

$$H = \left| \sum_{i=1}^{\text{KEYLTH}} \text{KEYVAL}_{(i)} \right| \pmod{\text{NEKISD}}$$

$$= \left| \sum_{i-1}^{2} \text{KEYVAL}_{(i)} \right| \pmod{188}$$

$$= \left| 7D7B_{16} \right| \pmod{188}$$

$$= 32{,}123 \pmod{188} = 16$$

If the number of actual key values exceeds 8,464, the value of NUMEKV must also exceed 8,464 so that 192 words (two sectors) will be used for the KIS directory; thus minimizing the number of KIS overflow blocks. Fewer KIS overflow blocks implies fewer mass-memory accesses in retrieving a record from the file.

On the other hand, if the actual number of key values is small and the estimate, NUMEKV, exceeds 8,464, the full 192 words will be used needlessly for the KIS directory; taking up 96 extra words of core each time the KIS directory is read in from mass memory. (The KIS directory for a given file is in core whenever an indexed file request is made for that file.)

In addition, the value of NUMEKV helps to determine the length of the KIS blocks since

$$\left\lfloor \sqrt{\text{NUMEKV}} \right\rfloor = \text{SRNEKV}$$

is the number of entries in each KIS block. Thus, too large a value of NUMEKV would result in unnecessarily long KIS blocks. Too small a value of NUMEKV results in KIS blocks that are too short, causing the creation of KIS overflow blocks. See Figures B-1 and B-2 for examples of key information segment structures dependent on the relationship between the number of expected key values and the number of actual key values.

STORAGE REQUIREMENTS FOR FILE STRUCTURE B

In this section, the equations for minimum/maximum storage limits are given so a particular file structure's storage requirements may be approximated. An example illustrates the calculation of the minimum/maximum storage limits. Finally, the equations are derived from the definition of the file structure given in Appendix A.

B.1 STORAGE LIMIT EQUATIONS

B.1.1 DEFINITIONS

Mnemonic	Description
MAXRL	Maximum record length of ith file
SRNEKV	Square root of the expected number of records with different key values (NUMEKV) of ith file
KEYLTH	Key length of ith file
RL	Length of record in ith file
NR	Number of records in ith file. Note that the number of records in a FIFO-linked file includes an extra record for each key stored
NWF_S	Number of words in ith sequential file
NWF_I	Number of words in ith indexed file
NWF_{MIN}	Minimum number of words in ith file
NWF_{MAX}	Maximum number of words in ith file
NF	Number of defined files
nf	Number of defined, but not released, files
NWFS	Number of words in the file structure
NUMPTR	Number of pointers in each KIS for an indexed file

B.1.2 ASSUMPTIONS

In some cases the storage limit equations are approximations that give a minimum and maximum range for the file structure. In other cases an exact computation of the file structure is given.

For storage efficiency, the maximum record length (MAXRL) should be an integer multiple of the record length (RL). Furthermore, the maximum record length should be of the form:

$$96 \cdot m - 3$$

Where: m is a positive integer. Relaxation of these restrictions may be investigated via Section B.3.3.3.

For calculation convenience the record length is assumed to be constant for any particular file. If this assumption is not true for a file, an average record length may be used. However, caution must be used so that this average record length does not violate any of the assumptions and restrictions on which the file structure is based.

B.1.3 SUMMARY OF STORAGE LIMIT EQUATIONS

From the derivations in Section B.3 and with the assumptions from Section B.1.2,

1. The number of words in the file record blocks is:

$$NWFRB' = (MAXRL + 3) \cdot \left\lceil \frac{NR \cdot RL}{MAXRL} \right\rceil$$

2. The minimum storage limit for an indexed file is:

$$NWF_{MIN} \geq \left(\frac{MAXRL + 98}{MAXRL + 95}\right)\left(RL \cdot NR\right) + SRNEKV^2 \, (KEYLTH + 2 \cdot NUMPTR)$$
$$+ 4 \cdot SRNEKV + 4$$

3. The maximum storage limit for an indexed file is:

$$NWF_{MAX} < \left(\frac{MAXRL + 3}{MAXRL}\right)\left(RL \cdot NR\right) + MAXRL + SRNEKV^2 \, (KEYLTH + 2 \cdot NUMPTR$$
$$+ SRNEKV(95 \cdot KEYLTH + 190 \cdot NUMPTR + 99) + 9507$$

4. The minimum storage limit for the file structure is:

$$NWFS \geq 96 + \frac{96 \cdot NF}{17} + \sum_{i=1}^{nf} NWF_{MIN}$$

5. The maximum storage limit for the file structure is:

$$\text{NWFS} \leq 96 + \frac{96(\text{NF} + 16)}{17} + \sum_{i=1}^{nf} \text{NWF}_{\text{MAX}}$$

B.2 EXAMPLE OF MINIMUM/MAXIMUM STORAGE LIMIT CALCULATIONS

The following is a hypothetical file structure

Where: NF = nf = 2 (one non-indexed and one indexed file)

File 1 contains	MAXRL	93 words
	RL	31 words
	NR	1880 records
File 2 contains	MAXRL	93 words
	RL	93 words
	NR	1880 records
	KEYLTH	8 words
	SRNEKV	50 words
	NUMPTR	1 word

$$\text{NWFRB}_{(1)} = \text{NWFRB}'_{(1)}$$

$$= (\text{MAXRL} + 3) \cdot \left\lceil \frac{\text{NR} \cdot \text{RL}}{\text{MAXRL}} \right\rceil$$

$$= 96 \cdot \left\lceil \frac{1880 \cdot 31}{93} \right\rceil$$

$$= 60{,}192$$

$$\text{NWF}_{\text{MIN}(2)} \geq \left(\frac{\text{MAXRL} + 98}{\text{MAXRL} + 95}\right) \text{RL} \cdot \text{NR} + \text{SRNEKV}^2 \, (\text{KEYLTH} + 2 \cdot \text{NUMPTR})$$

$$+ 4 \cdot \text{SRNEKV} + 4$$

$$\geq \left(\frac{93 + 98}{93 + 95}\right) (93 \cdot 1880) + 50^2 \, (8 + 2 \cdot 1) + 4(50) + 4$$

$$\geq 217{,}834 \text{ words}$$

$$NWF_{MAX(2)} < \left(\frac{MAXRL + 3}{MAXRL}\right)(RL \cdot NR) + MAXRL + SRNEKV^2\ (KEYLTH + 2 \cdot NUMPTR)$$

$$+ SRNEKV\ (95 \cdot KEYLTH + 190 \cdot NUMPTR + 99) + 9507$$

$$< \left(\frac{93 + 3}{93}\right)(93 \cdot 1880) + 93 + 50^2\ (8 + 2 \cdot 1) + 50\ (95 \cdot 8 + 190 \cdot 1 + 99)$$

$$+ 9507$$

$$< 282{,}530$$

$$NWFS \geq 96 + \frac{96 \cdot NF}{17} + \sum_{i=1}^{nf} NWF_{MIN}$$

$$\geq 96 + \frac{96 \cdot 2}{17} + NWF_{MIN(1)} + NWF_{MIN(2)}$$

$$\geq 277{,}151 \text{ words}$$

$$NWFS \leq 96 + \frac{96(NF + 16)}{17} + \sum_{i=1}^{nf} NWF_{MAX}$$

$$\leq 96 + \frac{96(2 + 16)}{17} + NWF_{MAX(1)} + NWF_{MAX(2)}$$

$$\leq 343{,}078 \text{ words}$$

Since the absolute minimum storage requirement ($NWFS_{AM}$) for this data structure would be:

$$NWFS_{AM} = \sum_{i=1}^{nf} RL_i \cdot NR_i$$

$$= RL_1 \cdot NR_1 + RL_2 \cdot NR_2$$

$$= 31 \cdot 1880 + 93 \cdot 1880$$

$$= 233{,}120 \text{ words}$$

The extra file storage needed for this example would be between 19% and 47% of the absolute minimum storage requirement.

B.3 DERIVATION OF STORAGE LIMIT EQUATIONS

B.3.1 DEFINITIONS

To the definitions given in Sections of B. 1. 1, the following are added:

Mnemonic	Description
NWFISD	Number of words in the FIS directory
FIDSIZ	FIS directory size in sectors
$NWFISB_{MIN}$	Minimum number of words in FIS blocks
$NWFISB_{MAX}$	Maximum number of words in FIS blocks
FIBSIZ	FIS block size in sectors
NFISB	Number of FIS blocks
NFFISB	Number of FISs in one FIS block
NWFIS	Number of words in a FIS
FRBSIZ	FRB size in sectors (see Section A. 2) of the ith file
NWFRB	Number of words in a FRB of ith file
NRFRB	Number of records in a FRB of the ith file
NFRB	Number of file record blocks in the ith file
$NWFRB_{MIN}$	Minimum number of words in FRBs of the ith file
$NWFRB_{MAX}$	Maximum number of words in FRBs of the ith file
NWFRB'	Desirable number of words in FRBs of the ith file
$NWKISD_{MIN}$	Minimum number of words in KIS directory of the ith file
$NWKISD_{MAX}$	Maximum number of words in KIS directory of the ith file
KIDSIZ	KIS directory size in sectors (see Section A. 3) of the ith file
$NWKISB_{MIN}$	Minimum number of words in KIS blocks of the ith file
$NWKISB_{MAX}$	Maximum number of words in KIS blocks of the ith file
NWKISB	Number of words in a KIS block of the ith file
KIBSIZ	KIS block size in sectors of the ith file
NKISB	Number of KIS blocks in the ith file
NWFISB	Number of words in FIS blocks
$NWFRB_i$	Number of words in FRBs of the ith file
$NWKISD_i$	Number of words in KIS directory of the ith file

Mnemonic	Description
$NWKISB_i$	Number of words in KIS blocks of the ith file
$NKKISB_i$	Number of KISs in one KIS block for the ith file
SRNEKV	$\lfloor \sqrt{NUMEKV} \rfloor$
NUMAKV	Number of actual key values.

B.3.2 INTEGER FUNCTION THEORY

If x is any real number, then

$\lfloor x \rfloor$ = the greatest integer less than or equal to x

$\lceil x \rceil$ = the least integer greater than or equal to x

Furthermore, it is noted that

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$$

and

$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n + m - 1}{m} \right\rfloor$$

where n and m are positive integers.

B.3.3 COMPUTATIONS

Computations are carried out to find minimum/maximum storage limit equations for the following, which is expressed as a function of a file's parameters:

- File information segment directory
- File information segment blocks
- File record blocks for the ith file
- Key information segment directory for the ith file
- Key information segment blocks for the ith file

B.3.3.1 COMPUTATION FOR FILE INFORMATION SEGMENT DIRECTORY

Computations are carried out for the storage size of the FIS directory. Note that NWFISD is a File Manager parameter and is defined to be 96.

$$\boxed{\text{NWFISD} = 96}$$

B.3.3.2 COMPUTATIONS FOR FILE INFORMATION SEGMENT BLOCKS

The number of words in all FIS blocks is computed as follows:

$$\begin{aligned}
\text{NWFISB} &= 288 \cdot \text{NFISB} \\
&= 288 \cdot \left\lceil \frac{\text{NF}}{\text{NFFISB}} \right\rceil \\
&= 288 \cdot \left\lceil \frac{\text{NF}}{\left\lfloor \frac{\text{NWFISB - 1}}{\text{NWFIS}} \right\rfloor} \right\rceil \\
&= 288 \cdot \left\lceil \frac{\text{NF}}{\left\lfloor \frac{288 - 1}{16} \right\rfloor} \right\rceil \\
&= 288 \cdot \left\lceil \frac{\text{NF}}{\left\lfloor \frac{287}{16} \right\rfloor} \right\rceil
\end{aligned}$$

$$\boxed{\text{NWFISB} = 288 \cdot \frac{\text{NF}}{17}}$$

B.3.3.3 COMPUTATIONS FOR FILE RECORD BLOCKS

Computations are carried out for three cases: the minimum, maximum, and desirable maximum storage limits for the file record blocks of the ith file. The results are obtained from the following information:

From Section A.3: $\text{FRBSIZ} = \left\lceil \frac{3 + \text{MAXRL}}{96} \right\rceil = \left\lfloor \frac{3 + \text{MAXRL} + 95}{96} \right\rfloor$

Since there are 96 words per sector: $\text{NWFRB} = \text{FRBSIZ} \cdot 96$

If it is assumed that the record length is constant for any particular file then (from Section A.2): $\text{NRFRB} = \frac{\text{NWFRB - 3}}{\text{RL}}$

Note that if the record length is not constant for any particular file, an average record length may be used, as long as there is a large number of records in the file.

By definition: $\text{NFRB} = \dfrac{\text{NR}}{\text{NRFRB}}$

MINIMUM LIMIT

$$
\begin{aligned}
\text{NWFRB}_{\text{MIN}} &= \text{NWFRB} \cdot \text{NFRB} \\
&= \text{NWFRB} \cdot \left\lceil \frac{\text{NR}}{\text{NRFRB}} \right\rceil \\
&\geq \text{NWFRB} \cdot \frac{\text{NR}}{\text{NRFRB}} \\
&= \text{NWFRB} \cdot \frac{\text{NR}}{\left\lfloor \dfrac{\text{NWFRB} - 3}{\text{RL}} \right\rfloor} \\
&\geq \text{NWFRB} \cdot \frac{\text{NR}}{\dfrac{\text{NWFRB} - 3}{\text{RL}}} \\
&\geq \left(\frac{\text{NWFRB}}{\text{NWFRB} - 3} \right) (\text{RL} \cdot \text{NR}) \\
&= \left(1 + \frac{3}{\text{NWFRB} - 3} \right) (\text{RL} \cdot \text{NR}) \\
&= \left(1 + \frac{3}{\text{FRBSIZ} \cdot 96 - 3} \right) (\text{RL} \cdot \text{NR}) \\
&= \left(1 + \frac{3}{\left\lfloor \dfrac{3 + \text{MAXRL} + 95}{96} \right\rfloor \cdot 96 - 3} \right) (\text{RL} \cdot \text{NR}) \\
&\geq \left(1 + \frac{3}{\left(\dfrac{3 + \text{MAXRL} + 95}{96} \right) \cdot 96 - 3} \right) (\text{RL} \cdot \text{NR}) \\
&= \left(1 + \frac{3}{\text{MAXRL} + 95} \right) (\text{RL} \cdot \text{NR})
\end{aligned}
$$

$$\boxed{\text{NWFRB}_{\text{MIN}} \geq \frac{\text{MAXRL} + 98}{\text{MAXRL} + 95} (\text{RL} \cdot \text{NR})}$$

MAXIMUM LIMIT

$$
\begin{aligned}
NWFRB_{MAX} &= NWFRB \cdot NFRB \\
&= NWFRB \cdot \left\lceil \frac{NR}{NRFRB} \right\rceil \\
&= NWFRB \cdot \left\lfloor \frac{NR + NRFRB - 1}{NRFRB} \right\rfloor \\
&\le NWFRB \cdot \left(\frac{NR + NRFRB - 1}{NRFRB} \right) \\
&= NWFRB \cdot \left(\frac{NR - 1}{NRFRB} + 1 \right) \\
&= NWFRB \cdot \left(\frac{NR - 1}{\left\lfloor \frac{NWFRB - 3}{RL} \right\rfloor} + 1 \right) \\
&= NWFRB \cdot \left(\frac{NR - 1}{\left\lceil \frac{NWFRB - RL - 2}{RL} \right\rceil} + 1 \right) \\
&\le NWFRB \cdot \left(\frac{NR - 1}{\left(\frac{NWFRB - RL - 2}{RL} \right)} + 1 \right) \\
&\le NWFRB \cdot \left(\frac{RL(NR - 1)}{NWFRB - RL - 2} + 1 \right) \\
&= \left(\frac{NWFRB}{NWFRB - RL - 2} \right) (RL \cdot NR - RL) + NWFRB \\
&= \left(1 + \frac{RL + 2}{NWFRB - RL - 2} \right) (RL \cdot NR - RL) + NWFRB \\
&= \left(1 + \frac{RL + 2}{FRBSIZ \cdot 96 - RL - 2} \right) (RL \cdot NR - RL) + FRBSIZ \cdot 96 \\
&= \left(1 + \frac{RL + 2}{\left\lfloor \frac{3 + MAXRL + 95}{96} \right\rfloor \cdot 96 - RL - 2} \right) (RL \cdot NR - RL) + \left\lfloor \frac{3 + MAXRL + 95}{96} \right\rfloor \\
&= \left(1 + \frac{RL + 2}{\left\lceil \frac{3 + MAXRL}{96} \right\rceil \cdot 96 - RL - 2} \right) (RL \cdot NR - RL) + \left\lfloor \frac{3 + MAXRL + 95}{96} \right\rfloor \cdot 96
\end{aligned}
$$

$$\leq \left(1 + \frac{RL + 2}{\left(\frac{3 + MAXRL}{96}\right) \cdot 96 - RL - 2}\right)(RL \cdot NR - RL) + \left(\frac{3 + MAXRL + 95}{96}\right) \cdot 96$$

$$= \left(1 + \frac{RL + 2}{MAXRL + 1 - RL}\right)(RL \cdot NR - RL) + MAXRL + 98$$

$$\boxed{NWFRB_{MAX} \leq \left(\frac{MAXRL + 3}{MAXRL + 1 - RL}\right)(RL \cdot NR - RL) + MAXRL + 98}$$

DESIRABLE MAXIMUM LIMIT

The maximum limit provides for the likelihood of an extremely bad choice for the record length parameter (RL). However, if this parameter is chosen carefully, storage will be conserved and a more desirable maximum limit can be computed and used to approximate the storage requirement.

The record length should be chosen such that the term

$$\frac{NWFRB - 3}{RL}$$

is (or is slightly less than) an integer. This implies that the record length is a factor of (NWFRB - 3), or

$$\begin{aligned} RL \cdot n &= NWFRB - 3 \\ &= FRBSIZ \cdot 96 - 3 \\ &= \left\lfloor \frac{3 + MAXRL + 95}{96} \right\rfloor \cdot 96 - 3 \\ &= \left\lceil \frac{3 + MAXRL}{96} \right\rceil \cdot 96 - 3 \end{aligned}$$

where n is a positive integer.

If the maximum record length (MAXRL) is of the form: $MAXRL = 96 \cdot m - 3$

where m is a positive integer, then $RL \cdot n = 96 \cdot m - 3$

$$= MAXRL$$

Thus, if the record length is a factor of the maximum length, storage space is conserved and the following desirable number of words can be utilized:

$$\begin{aligned}
NWFRB' &= (MAXRL + 3) \cdot NFRB \\
&= (MAXRL + 3) \cdot \left\lceil \frac{NR}{NRFRB} \right\rceil \\
&= (MAXRL + 3) \cdot \left\lceil \frac{NR}{\left\lfloor \frac{FRBSIZ \cdot 96 - 3}{RL} \right\rfloor} \right\rceil \\
&= (MAXRL + 3) \cdot \left\lceil \frac{NR}{\left\lfloor \frac{\left\lceil \frac{3 + MAXRL}{96} \right\rceil \cdot 96 - 3}{RL} \right\rfloor} \right\rceil \\
&= (MAXRL + 3) \cdot \left\lceil \frac{NR \cdot RL}{MAXRL} \right\rceil
\end{aligned}$$

B.3.3.4 COMPUTATIONS FOR KIS DIRECTORY

The number of words in the KIS directory is the number of sectors in the KIS directory, KIDSIZ, multiplied by 96.

$$NWKISD = KIDSIZ \cdot 96$$

The number of sectors in the directory must be large enough to accommodate four header words plus one word for a pointer to each KIS block needed for the file.

The number of KIS blocks needed for the file is NKISB, thus:

$$KIDSIZ = \left\lceil \frac{4 + NKISB}{96} \right\rceil \text{ and } NWKISD = \left\lceil \frac{4 + NKISB}{96} \right\rceil \cdot 96$$

The File Manager is designed so that the number of KISs in each KIS block is equal to the number of KIS blocks; that is,

$$NKKISB = NKISB$$

Also by design of the File Manager,

$$\begin{aligned}
NUMEKV &= NKISB \cdot NKKISB \\
&= NKISB^2
\end{aligned}$$

or,

$$NKISB = \left\lfloor \sqrt{NUMEKV} \right\rfloor = SRNEKV$$

Therefore, $NWKISD = \left\lceil \frac{4 + SRNEKV}{96} \right\rceil \cdot 96$

But, $\frac{SRNEKV}{96} \le \left\lceil \frac{4 + SRNEKV}{96} \right\rceil = \left\lfloor \frac{4 + SRNEKV + 95}{96} \right\rfloor \le \frac{99 + SRNEKV}{96}$

Therefore,

$$\boxed{\begin{aligned} NWKISD_{MIN} &= SRNEKV + 4 \le NWKISD \le SRNEKV + 99 \\ &= NWKISD_{MAX} \end{aligned}}$$

B.3.3.5 COMPUTATIONS FOR KEY INFORMATION SEGMENT BLOCKS

From Section A.4, $NWKISB = \left\lceil \frac{3 + (KEYLTH + 2 \cdot NUMPTR) \cdot SRNEKV}{96} \right\rceil \cdot 96$

Thus, the minimum length for a KIS block occurs when both KEYLTH and NUMEKV are small and the file is not indexed-linked.

Suppose	KEYLTH	= 1
	NUMPTR	= 1
	NUMEKV	= 2
then,	NWKISB	= 96

The maximum length for a KIS block occurs when KEYLTH and NUMEKV are assigned their maximum values and the file is FIFO index-linked.

If	KEYLTH	= 63
	NUMPTR	= 2
	NUMEKV	= 32,767
then,	NWKISB	= 12,000

The expected number of KIS blocks is dependent on the expected number of key values (NUMEKV) and on the actual number of key values (NUMAKV). The relationship for the case NUMAKV = NUMEKV is shown in Table B-1. The number of KIS overflow blocks depends on how NUMEKV relates to the actual number of key values and on whether or not the key values scatter uniformly.

Table B-1. Number of KIS Blocks as a Function of Number of Expected Key Values

SRNEKV = NUMBER OF KISs IN EACH KIS BLOCK	NUMAKV = NUMEKV*	EXPECTED NUMBER OF KIS BLOCKS	NEKISD = MAXIMUM NUMBER OF ENTRIES IN KIS DIRECTORY	LENGTH OF KIS DIRECTORY	LENGTH OF KIS BLOCK FOR FILE WITH KEYLTH = 4 AND NO INDEX-LINKING
1	1	1	92	96	96
1	2	2	92	96	96
1	3	3	92	96	96
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
9	91	91	92	96	96
9	92	92	92	96	96
9	93	92	92	96	96
9	94	92	92	96	96
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
92	8,464	92	92	96	384
92	8,465	93	188	192	384
92	8,466	94	188	192	384
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
92	8,559	187	188	192	384
92	8,560	188	188	192	384
92	8,561	188	188	192	384
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
181	32,766	188	188	192	768
181	32,767	188	188	192	768

*NUMAKV = Number of actual key values
NUMEKV = Number of expected key values

In the following discussion we will assume uniform scattering. Let NUMAKV be the number of actual key values and NEKISD be the maximum number of entries in the KIS directory, then, NUMAKV/NEKISD is the number of expected key values with the same scatter code. Since SRNEKV KISs can appear in one KIS block, when NUMAKV/NEKISD is less than or equal to SRNEKV, the number of KIS overflow blocks is zero. Let NEKISO denote the number of KIS overflow blocks. Then for NUMAKV/NEKISD ≤ SRNEKV, we have

$$\text{NEKISO} = 0$$

For NUMAKV/NEKISD > SRNEKV, we have

$$\text{NEKISO} = \left(\left\lfloor\frac{\text{NUMAKV}}{\text{NEKISD} \cdot \text{SRNEKV}}\right\rfloor - 1\right) \cdot \text{NEKISD} + \text{NUMAKV} \pmod{\text{NEKISD}}$$

Note that NUMAKV/NEKISD is the expected number of keys per scatter code; SRNEKV is the number of keys per KIS block; and NEKISD is the number of scatter codes. Also, when NUMAKV/NEKISD is less than SRNEKV, there will be one KIS block for each of the NEKISD scatter codes.

See Table B-2 for some examples of the relationship between the expected number of KIS overflow blocks and key values and the actual number of key values.

Table B-2. Expected Number of KIS Overflow Blocks as Related to Expected and Actual Key Values

NUMAKV	NUMEKV	NEKISD	SRNEKV = NUMBER OF KEYS PER KIS BLOCK	EXPECTED NUMBER OF KIS BLOCKS	EXPECTED NUMBER OF KIS OVERFLOW BLOCKS
1	1	92	1	1	0
92	92	92	9	92	0
8,464	8,464	92	92	92	0
32,767	32,767	188	181	188	0
100	1	92	1	92	8
100	16	92	4	92	8
100	8,464	92	92	92	0
9,000	1	92	1	92	8,908
9,000	8,464	92	9	92	92
9,000	32,767	188	181	188	0

B.3.4 EQUATIONS

The equations for calculating the minimum/maximum storage limits for non-indexed and indexed files, as well as the total minimum storage limits for the file structure are given.

B.3.4.1 STORAGE LIMITS FOR NON-INDEXED FILE

From Appendix A:

$$NWF_S = NWFRB_i$$

and from Section B.3.3.3 (assuming the desired maximum limit):

$$\boxed{NWF_S = (MAXRL + 3) \cdot \left\lceil \frac{NR \cdot RL}{MAXRL} \right\rceil}$$

B.3.4.2 STORAGE LIMITS FOR INDEXED FILE

From Appendix A:

$$NWF_I = NWFRB_i + NWFISD_i + NWKISB_i$$

and from Section B.3.3.3 (assuming the desired maximum limit) and Sections B.3.3.4 and B.3.3.5 (assuming NUMAKV = NUMEKV):

$$\boxed{\begin{aligned} NWF_I = (MAXRL + 3) \cdot \left\lceil \frac{NR \cdot RL}{MAXRL} \right\rceil + \left\lceil \frac{4 + SRNEKV}{96} \right\rceil \\ 96 + SRNEKV \cdot 96 \cdot \\ \left\lceil \frac{SRNEKV \cdot (KEYLTH + 2 \cdot NUMPTR) + 3}{96} \right\rceil \end{aligned}}$$

B.3.4.3 TOTAL STORAGE LIMITS FOR FILE STRUCTURE

From Appendix A:

$$NWFS = NWFISD + NWFISB + \sum_{i=1}^{nf} NWF_i$$

and from Sections B.3.3.1 and B.3.3.2:

$$\boxed{NWFS = 96 + 288 \cdot \left\lceil \frac{NF}{17} \right\rceil + \sum_{i=1}^{nf} NWF_i}$$

ACCESS RATES FOR FILE STRUCTURE C

(Refer to Section 4.1.1 for definition of symbols.)

C.1 ACCESS EQUATIONS FOR STORAGE/RETRIEVAL METHODS

The access equations for the sequential, indexed, and direct storage/retrieval methods, as well as the initial and terminal usage of the files are calculated using the definition of the file structure given in Appendix A.

C.1.1 ACCESSES FOR SEQUENTIAL METHOD

STORE

A sequential store requires one access for storing the record, plus one extra access for storing a file record block pointer, if the record is the first record in the file record block. Thus, to store all the records in a file record block, one more store than the total number of records in the file record block is required. The number of accesses for storing a sequential record is minimized when all the records in a given file record block are stored in loop. Within such a loop, the number of accesses for storing a sequential record is computed as follows:

$$
\begin{aligned}
NA_{SS} &= \frac{NRFRB + 1}{NRFRB} \\
&= 1 + \frac{1}{NRFRB} \\
&= 1 + \frac{1}{\left\lfloor \dfrac{NWFRB - 3}{RL} \right\rfloor} \\
&= 1 + \frac{1}{\left\lfloor \dfrac{FRBSIZ \cdot 96 - 3}{RL} \right\rfloor} \\
&= 1 + \frac{1}{\left\lfloor \dfrac{\left\lceil \dfrac{3 + MAXRL}{96} \right\rceil \cdot 96 - 3}{RL} \right\rfloor}
\end{aligned}
$$

If the assumptions in Section B. 2. 2 are true, then

$$NA_{SS} = 1 + \frac{1}{\left\lfloor \frac{MAXRL}{RL} \right\rfloor}$$

$$\boxed{NA_{SS} \doteq 1 + \frac{RL}{MAXRL}}$$

Note that if the record length (RL) equals the maximum record length (MAXRL), two accesses are required for each sequential store. However, as the limit of RL/MAXRL approaches zero, only one access is required.

<u>RETRIEVE</u>

A sequential retrieve requires one access for retrieving the next sequential record, plus one extra access if the record is being removed. Therefore the number of accesses for the next sequential retrieve is given by:

$$\boxed{NA_{NSR} = 1 + f(ro)}$$

Note that the average number of accesses to retrieve any record in a sequential file is given by:

$$\boxed{NA_{ASR} = \frac{NR}{2} + f(ro)}$$

C.1.2 ACCESSES FOR THE INDEXED METHOD

<u>STORE</u>

An indexed store requires one access for retrieving the desired key information segment block and one access to update it with the proper information, plus the number of accesses required for a sequential store (Section C. 1. 1) to store the record. Therefore the number of accesses for storing one indexed record as a part of a loop to store all the indexed records in one file record block is given by

$$\boxed{NA_{IS} = 3 + \frac{RL}{MAXRL}}$$

RETRIEVE

An indexed retrieve requires one access for retrieving the desired key information segment block and one access for retrieving the record, plus two extra accesses if the record is being removed. Therefore the number of accesses for an indexed retrieve is given by:

$$\boxed{NA_{IR} = 2 + 2 \cdot f(ro)}$$

C.1.3 ACCESSES FOR THE DIRECT METHOD

STORE

A direct store requires one access for updating the record. Therefore the number of accesses for a direct store is given by:

$$\boxed{NA_{DS} = 1}$$

RETRIEVE

A direct retrieve requires one access for retrieving the record, plus one extra access if the record is being removed. Therefore the number of accesses for a direct retrieve is given by:

$$\boxed{NA_{DR} = 1 + f(ro)}$$

C.1.4 ACCESSES DUE TO THE INITIAL USAGE OF A FILE

NON-INDEXED

If there have been no accesses via file requests of the file structure for a period of time, the next access is designated an initial usage of the file structure. This requires one extra access to read in the file information segment directory. Furthermore, if there have been no accesses via file requests of a particular file for a period of time, the next access to this file is designated an initial usage of the file. Assuming that the file numbers scatter uniformly, the number of FISs with the same scatter code is given by:

$$\text{NFISSS} = \left\lceil \frac{\text{NF}}{\left(\frac{\text{NWFISD - 2}}{2}\right)} \right\rceil$$

Since NWFISD equals 96 (see Section B.3.3.1), then

$$\text{NFISSS} = \left\lceil \frac{\text{NF}}{47} \right\rceil$$

On the average, the expected number of FISs to be accessed to find the FIS corresponding to a given file number is given by

$$\left\lceil \frac{1}{2} \cdot \left\lceil \frac{\text{NF}}{47} \right\rceil \right\rceil \leq \left\lceil \frac{1}{2} \cdot \frac{\text{NF}}{47} \right\rceil = \left\lceil \frac{\text{NF}}{94} \right\rceil$$

This gives an upper bound on the number of file information segment blocks that must be accessed to read in a given file information segment.

Therefore, the expected number of accesses for an initial usage of a non-indexed file is given by:

$$\boxed{\text{NA}_{\text{IUNF}} \leq 1 + \left\lceil \frac{\text{NF}}{94} \right\rceil}$$

INDEXED

If there have been no accesses via indexed file requests of a particular indexed file for a period of time, the next indexed access to this file is designated an initial indexed usage of the file. This requires one extra access to read in the key information segment directory. Furthermore, if the indexed file has not been accessed for a period of time by non-indexed methods, extra accesses are also required as in the non-indexed case. Therefore, the number of accesses for an initial usage of an indexed file is given by:

$$\boxed{\text{NA}_{\text{IUIF}} \leq 2 + \left\lceil \frac{\text{NF}}{94} \right\rceil}$$

Note: In this section it has been assumed that the user has not followed the file access time optimization procedure described in Section 4.3. If the user has followed this optimization procedure, all the files with one scatter code will be stored in one FIS block. If the procedure has been followed and if the number of files is less than or equal to

$$\left(\begin{array}{l}\text{Maximum number of scatter codes}\\ \text{represented in FIS directory}\end{array}\right) \cdot \left(\begin{array}{l}\text{Maximum number of FISs that can be}\\ \text{contained in one FIS}\end{array}\right)$$

$$= 94 \cdot 18 = 1,598$$

then the expression

$$\left\lceil \frac{NF}{94} \right\rceil$$

in the equations of this section can be replaced by the value 1.

C.1.5 ACCESSES DUE TO THE TERMINAL USAGE OF A FILE

NON-INDEXED

If the file structure is not accessed for a period of time, one extra access is required to write out the file information segment directory (if it has changed). Furthermore, if a particular file is not used for a period of time, one extra access is required to write out the file information segment (if it has changed). Therefore the number of accesses for the terminal usage of a non-indexed file is given by

$$\boxed{NA_{TUNF} \leq 2}$$

INDEXED

If a particular indexed file is not accessed for a period of time, one extra access is required to write out the key information segment directory (if it has changed). Furthermore, if the indexed file has not been accessed by non-indexed methods, extra accesses are also required as in the non-indexed case. Therefore the number of accesses for the terminal usage of an indexed file is given by:

$$\boxed{NA_{TUIF} \leq 3}$$

C.2 DISK/DRUM AVERAGE TRANSFER RATE

C.2.1 DISK AVERAGE TRANSFER RATE

The average transfer rate of the 853 disk is quite complex when word addressing is utilized. However, approximations can be made if the number of words that are to be read or written is small (e.g., $RL \leq 93$). With this assumption, the average transfer rate for a disk read or write is given by *:

$$TR_{DKR} \approx \overline{SEEK} + \overline{LAT}_{DK} + LAT_{DK}\left(\frac{RL - 1}{96}\right)$$

$$\approx 110 + 13 + 25\left(\frac{RL - 1}{96}\right)$$

$$\boxed{TR_{DKR} \approx 123 + \frac{25}{96}(RL - 1) \text{ milliseconds/access}}$$

and

$$TR_{DKW} \approx \overline{SEEK} + \overline{LAT}_{DK} + LAT_{DK} + 2 \cdot LAT_{DK}\left(\frac{RL - 1}{96}\right)$$

$$\approx 110 + 13 + 25 + 2.25\left(\frac{RL - 1}{96}\right)$$

$$\boxed{TR_{DKW} \approx 148 + \frac{50}{96}(RL - 1) \text{ milliseconds/access}}$$

Note that the above assumes that the software overhead is negligible.

C.2.2 DRUM AVERAGE TRANSFER RATE

The average transfer rate of the 1751 drum is the sum of the average latency and the word transfer times. Therefore:

$$TR_{DM} = \overline{LAT}_{DM} + TWR_{DM} \cdot RL$$

$$\boxed{TR_{DM} = 8 + .008 \cdot RL \text{ milliseconds}}$$

*The derivation of these approximations is not proven here because of space considerations. For larger records ($RL > 93$), the transfer rates may be modestly or substantially increased.

FIS, FRB, KIS, AND FILE SPACE POOL DUMPS D

One sequential and one indexed file are defined in the FORTRAN code in Figure D-1. Three records are stored into the sequential file, file number 256. Each data word contains the record number for each sequential record. Four records are stored into the indexed file, file number 97. Each data word of the record with the key value AA_{16} contains the value A_{16} and each data word of the record with the key value BB_{16} contains the value B_{16}. Similarly, the record with the key value CC_{16} is filled with C_{16}s and the record with the key value DD_{16} is filled with D_{16}s.

```
      PROGRAM EXAMPL
      DIMENSION IREQBF{12}¬IRECBF{93}¬IOBUF{58}¬IRECPT{2}
C DEFINE FILE NUMBER 256 {=$100}
      IFLNUM=256
C LOGICAL UNIT 8 IS THE DISK
      LU=8
C TO OPTIMIZE USE OF FILE SPACE IN FILE BLOCKS¬ LET MAXRL=93{=96*1-3}
      MAXRL=93
      CALL DEFFIL{IFLNUM¬MAXRL¬LU¬IREQBF¬IREQID}
C CHECK FOR ERRORS
      IF {IREQID.LT.0} GO TO 5000
C DEFINE FILE NUMBER 97
      IFLNUM=97
      MAXRL=93
      CALL DEFFIL {IFLNUM¬MAXRL¬LU¬IREQBF¬IREQID}
C CHECK FOR ERRORS
      IF {IREQID.LT.0} GO TO 5000
C DEFINE FILE 97 AS AN INDEXED FILE WITH A ONE-WORD KEY AND
C 400 EXPECTED KEY VALUES
      KEYLTH=1
      NUMEKV=400
      CALL DEFIDX{IFLNUM¬NUMEKV¬KEYLTH¬LU¬IREQBF¬IREQID}
C CHECK FOR ERRORS
      IF {IREQID.LT.0} GO TO 5000
C STORE 3 RECORDS INTO SEQUENTIAL FILE {FILE NUMBER 256}
      IFLNUM=256
C EACH RECORD HAS 31 WORDS
      DO 100 IREC=1¬3
C LET CONTENTS OF EACH DATA WORD OF THE RECORD BE THE RECORD NUMBER.
C NOTICE THAT NOTHING IS STORED IN THE FIRST WORD OF A RECORD¬AS THIS
```

Figure D-1. FORTRAN Code Example (Sheet 1)

```
C WORD IS RESERVED FOR USE BY THE FILE MANAGER.
      DO 50 IWORD=2,31
   50 IRECBF{IWORD}=IREC
      CALL STOSEQ{IFLNUM,IRECPT,IRECBF,31,IREQBF,IREQID}
C CHECK FOR ERRORS
      IF {IREQID.LT.0} GO TO 5000
  100 CONTINUE
C STORE FOUR RECORDS INTO FILE 97, WITH KEYS=$AA,$BB,$CC, AND $DD,
C RESPECTIVELY.  LET EACH DATA WORD OF THE RECORD WITH KEY VALUE $AA
C CONTAIN THE VALUE $A.  LET EACH DATA WORD OF THE RECORD WITH KEY
C VALUE $BB CONTAIN THE VALUE $B.  SIMILARLY, THE RECORD WITH KEY
C VALUE $CC IS TO BE FILLED WITH $C≠S, AND THE RECORD WITH KEY VALUE
C $DD IS TO BE FILLED WITH $D≠S.
C INITIALIZE KEY VALUE AND DATA WORD VALUE.
      KEYVAL=$AA
      IDATA= $A
      IFLNUM=97
      DO 200 IREC=1,4
      DO 150 IWORD=2,93
  150 IRECBF{IWORD}=IDATA
      CALL STOIDX {IFLNUM,KEYVAL,IRECPT,IRECBF,93,IREQBF,IREQID}
C CHECK FOR ERRORS
      IF {IREQID.LT.0} GO TO 5000
C INCREMENT KEY VALUE AND DATA WORD VALUE
      KEYVAL=KEYVAL+$11
      IDATA=IDATA+1
  200 CONTINUE
      GO TO 5010
C PRINT ERROR MESSAGE
 5000 CALL SETBFR{IOBUF,58}
      WRITE {4,6000} IFLNUM,IREQID
 5010 CONTINUE
      CALL RELESE{EXAMPL}
 6000 FORMAT{5HFILE ,I5,8H ERROR $,$4}
      END
```

Figure D-1. FORTRAN Code Example (Sheet 2)

After execution of the program, the FIS directory is dumped (as shown in Figure D-2). The FILMGR SYSDAT core location FIDSEC is dumped to determine the location of the FIS directory. The value of FIDSEC is $2EA_{16}$, therefore mass memory sector $2EA_{16}$ is dumped to obtain the FIS directory.

The first word of the FIS directory contains $2EB_{16}$ which points to the first FIS block. The scatter code for file 256 is 256 (mod 47) = 21. The 21st two-word entry in the FIS directory is $2EB_{16}$, 1, indicating that the FIS for file 256 is to be found in sector $2EB_{16}$, word 1. The scatter code for file 97 is 97(mod 47) = 3. The third entry in the FIS directory indicates that the FIS for file 97 is to be found at sector $2EB_{16}$, word 11_{16}.

```
02EA
   02EB  000A  0000  0000  0000  0000  02EB  0011
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  02EB  0001  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
   0000  0000  0000  0000  0000  0000  0000  0000
```

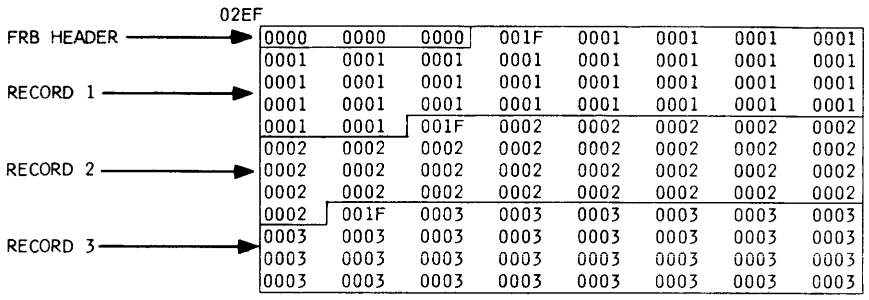
Figure D-2. Example of FIS Directory

The 288-word FIS block containing the FIS for file 256 and the FIS for file 97 is shown in Figure D-3. The FIS header word contains a zero that indicates there are no more FIS blocks. The FIS for file 256 is found in words 1_{16} through 10_{16} of the FIS block. Words 11_{16} through 20_{16} contain the FIS for file 97.

```
                        02EB
FIS HEADER WORD ------>   0000  0000  0000  0100  02EF  0003  02EF  0060
FIS FOR FILE 256------>   0000  0000  0000  0000  0000  0000  0001  0001
                          C008  0000  0000  0061  02F0  0001  02F6  0060
FIS FOR FILE 97------->   02FF  0001  0001  0001  0190  0000  0004  0001
                          C408  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                        02EC
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                        02ED
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
                          0000  0000  0000  0000  0000  0000  0000  0000
```

Figure D-3. FIS Block Example

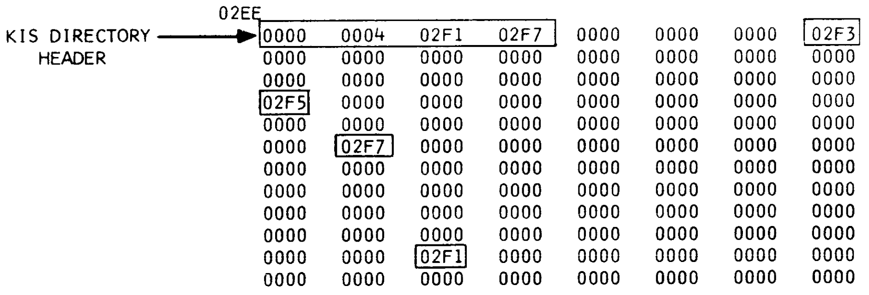
Words 3, 4, and 5 of the FIS for file 256 indicate that the file contents for this file are in sector $2EF_{16}$. Section $2EF_{16}$ is shown in Figure D-4. The three-word FRB header indicates that this is the first FRB and that there are no more FRBs. The remainder of the sector contains the three records stored in the file and the first word of each record contains the record length.



```
               02EF
FRB HEADER ------->  0000  0000  0000  001F  0001  0001  0001  0001
                     0001  0001  0001  0001  0001  0001  0001  0001
RECORD 1 --------->  0001  0001  0001  0001  0001  0001  0001  0001
                     0001  0001  0001  0001  0001  0001  0001  0001
                     0001  0001  001F  0002  0002  0002  0002  0002
                     0002  0002  0002  0002  0002  0002  0002  0002
RECORD 2 --------->  0002  0002  0002  0002  0002  0002  0002  0002
                     0002  0002  0002  0002  0002  0002  0002  0002
                     0002  001F  0003  0003  0003  0003  0003  0003
RECORD 3 --------->  0003  0003  0003  0003  0003  0003  0003  0003
                     0003  0003  0003  0003  0003  0003  0003  0003
                     0003  0003  0003  0003  0003  0003  0003  0003
```

Figure D-4. FRB Example Sequential File

File 97 is an indexed file. To find a record corresponding to a given key value, the KIS directory must be examined. Word 7 of the FIS for file 97 indicates that the KIS directory is located at sector $2EE_{16}$ (refer to Figure D-3). The KIS directory is shown in Figure D-5. The directory header shows that there are four linked KIS blocks, the first in sector $2F1_{16}$ and the last in sector $2F7_{16}$.



```
                     02EE
KIS DIRECTORY ---->  0000  0004  02F1  02F7  0000  0000  0000  02F3
   HEADER            0000  0000  0000  0000  0000  0000  0000  0000
                     0000  0000  0000  0000  0000  0000  0000  0000
                     02F5  0000  0000  0000  0000  0000  0000  0000
                     0000  0000  0000  0000  0000  0000  0000  0000
                     0000  02F7  0000  0000  0000  0000  0000  0000
                     0000  0000  0000  0000  0000  0000  0000  0000
                     0000  0000  0000  0000  0000  0000  0000  0000
                     0000  0000  0000  0000  0000  0000  0000  0000
                     0000  0000  0000  0000  0000  0000  0000  0000
                     0000  0000  02F1  0000  0000  0000  0000  0000
                     0000  0000  0000  0000  0000  0000  0000  0000
```

Figure D-5. KIS Directory for File 97, A Sample File

Figure D-6 shows four entries in the KIS directory and the key value, scatter code, and KIS block pointer for each entry. The first KIS directory entry points to the KIS block with scatter code zero, as opposed to the first FIS directory entry, which points to the first FIS with scatter code one.

KEY VALUE	K = DECIMAL EQUIVALENT OF KEY VALUE	SCATTER CODE = K(MOD 92)	KIS POINTER (FROM FIGURE D-5)
AA_{16}	170	78	$2F1_{16}$
BB_{16}	187	3	$2F3_{16}$
CC_{16}	204	20	$2F5_{16}$
DD_{16}	221	37	$2F7_{16}$

Figure D-6. Key Values, Scatter Codes, and Corresponding KIS Pointers

The KIS block for key value AA_{16} is located in sector $2F1_{16}$, as shown in Figure D-7. The KIS block header indicates that this is the first KIS block and that there are no KIS overflow blocks. There is one KIS in this KIS block, which is located in words 3 through 5 of the sector. It shows that the record with key value AA_{16} is stored in sector 2F0, starting at word 3. The record is also shown in Figure D-7.

```
                          02F0
FRB HEADER ------------> 0000  02F2  0001 | 005D  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A
RECORD CORRE-  --------> 000A  000A  000A  000A  000A  000A  000A  000A
SPONDING TO KEY          000A  000A  000A  000A  000A  000A  000A  000A
VALUE AA16               000A  000A  000A  000A  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A
                         000A  000A  000A  000A  000A  000A  000A  000A


                                                    ---------- KIS FOR KEY
                                                   |           VALUE AA16
                          02F1                     v
KIS BLOCK -------------> 0000  0000  0001 | 02F0  0003  00AA  0000  0000
HEADER                   0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
                         0000  0000  0000  0000  0000  0000  0000  0000
```

Figure D-7. KIS and Corresponding Record

Similarly, the KIS and the corresponding record for each of the key values BB_{16}, CC_{16}, and DD_{16} are shown in Figure D-8.

02F2

	02F0	02F4	0001	005D	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
FRB →	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B
	000B	000B	000B	000B	000B	000B	000B	000B

02F5 KIS↓

02F3	0000	0001	02F4	0003	00CC	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

02F3 KIS↓

02F1	0000	0001	02F2	0003	00BB	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

02F6

	02F4	0000	0000	005D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
FRB →	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D
	000D	000D	000D	000D	000D	000D	000D	000D

02F4

	02F2	02F6	0001	005D	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
FRB →	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C
	000C	000C	000C	000C	000C	000C	000C	000C

02F7 KIS↓

02F5	0000	0001	02F6	0003	00DD	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

Figure D-8. Three KISs and Their Corresponding Records

After execution of the program shown in Figure D-1, the file space list and pool are dumped. In the system used in this example, the file space list is composed of a block of 1-sector, 2-sector, and 3-sector segments. When dumped, the file space list is empty. The SYSDAT FILMGR word FSPOOL contains $2F8_{16}$; the first three words of this sector are shown below:

```
02F8
    0000    0000    3FF2
```

A diagram of the file space pool is shown in Figure D-9. This figure may be compared with the other file space pool example shown in Figure 2-1.

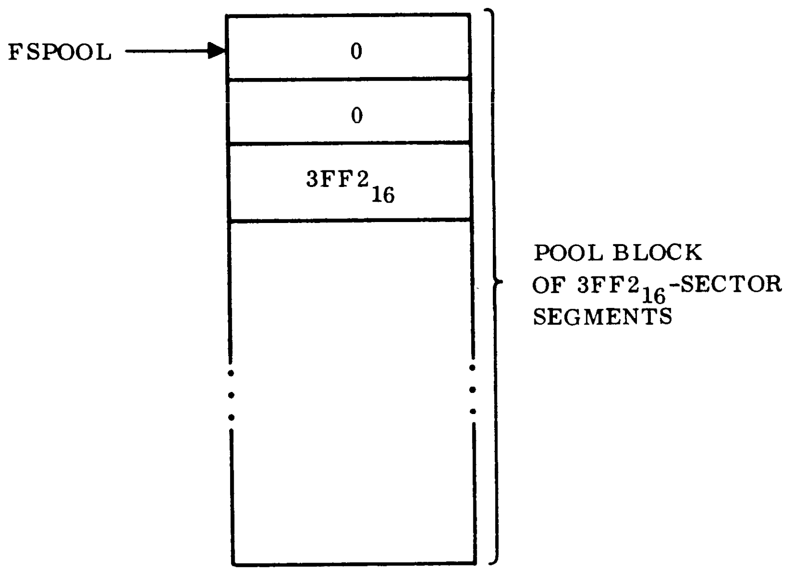


Figure D-9. File Space Pool

The first zero pointer indicates that there are no other segments of this length in the pool. The second zero pointer indicates that there are no pool blocks of segments having a greater length. The third word indicates that the length of this segment is $3FF2_{16}$ sectors.

FILE STRUCTURE ILLUSTRATIONS E

The following illustrations of the file structure are given:

E.1 FILE STRUCTURE FOR STORAGE/RETRIEVAL METHODS

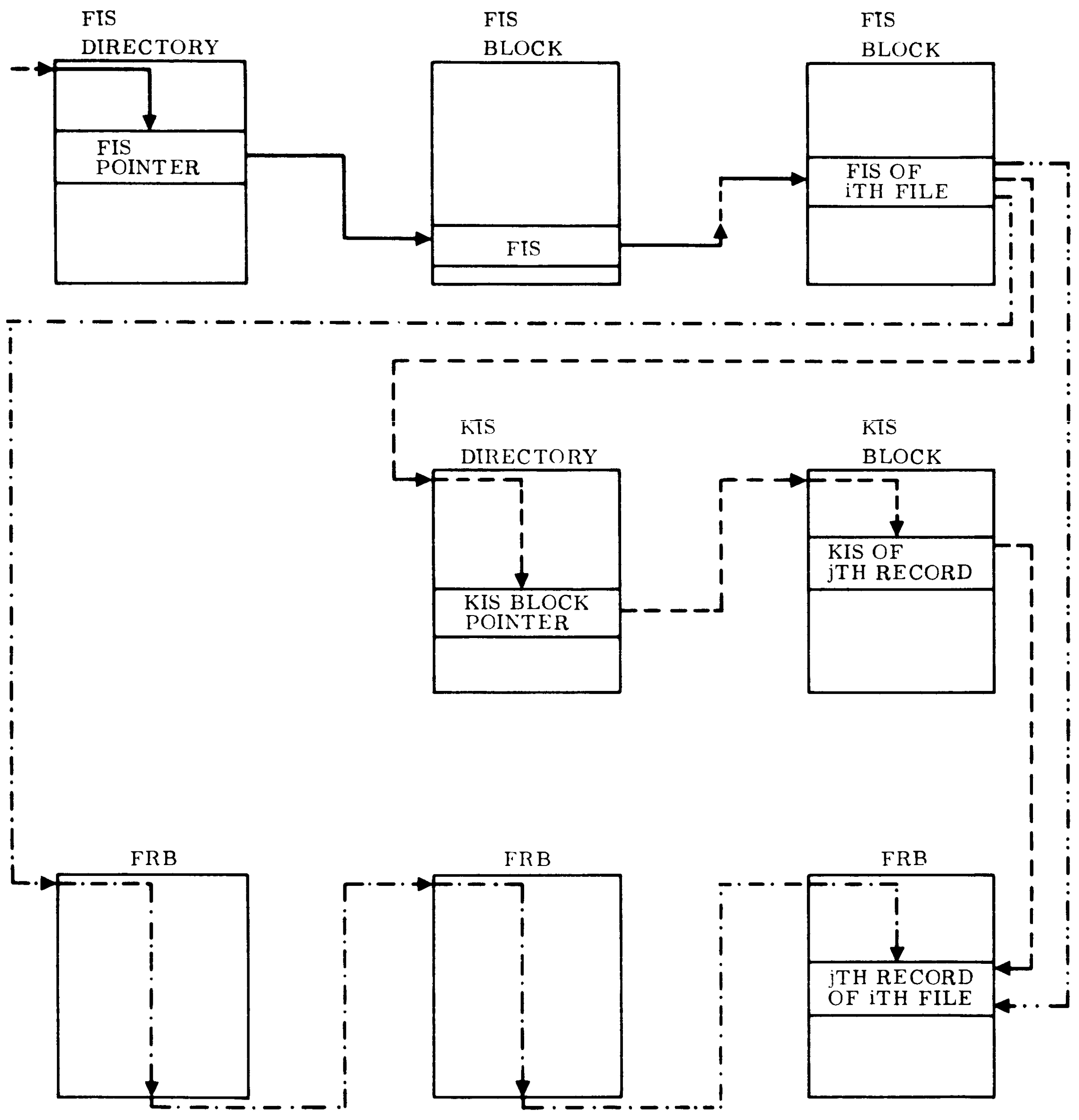
The flow logic for sequential, indexed, and direct storage/retrieval are illustrated in Figure E-1. Note that all three share a common path, i.e., the flow logic through the FIS directory and the FIS blocks. Once the file information segment is found, each proceeds on its separate path until the record is retrieved.

E.2 FILE STRUCTURE FOR INDEXED-LINKED WITH LIFO-LINKING

The file structure for indexed-linked with LIFO-linking is illustrated in Figure E-2. Note that records 2, 6, and 8 have the same key value (B), records 1 and 5 have the same key value (C), records 3 and 4 have the same key value (D), and record 7 has a unique key value (A).

E.3 FILE STRUCTURE FOR INDEXED-ORDERED

The file structure for indexed-ordered files is illustrated in Figure E-3. This example assumes that the number of expected key values is nine; therefore, there are three KIS blocks with each KIS block having three key information segments. In the figure five records have been stored. Note that each key information segment is stored in a particular KIS block and is ordered within that block by its key value.



Notes: 1. ——→ Common flow logic
2. —·—·→ Sequential flow logic
3. — — —→ Indexed flow logic
4. —··—··→ Direct flow logic
5. ············ Repeated logic

Figure E-1. File Structure Flow Logic for Storage/Retrieval Methods

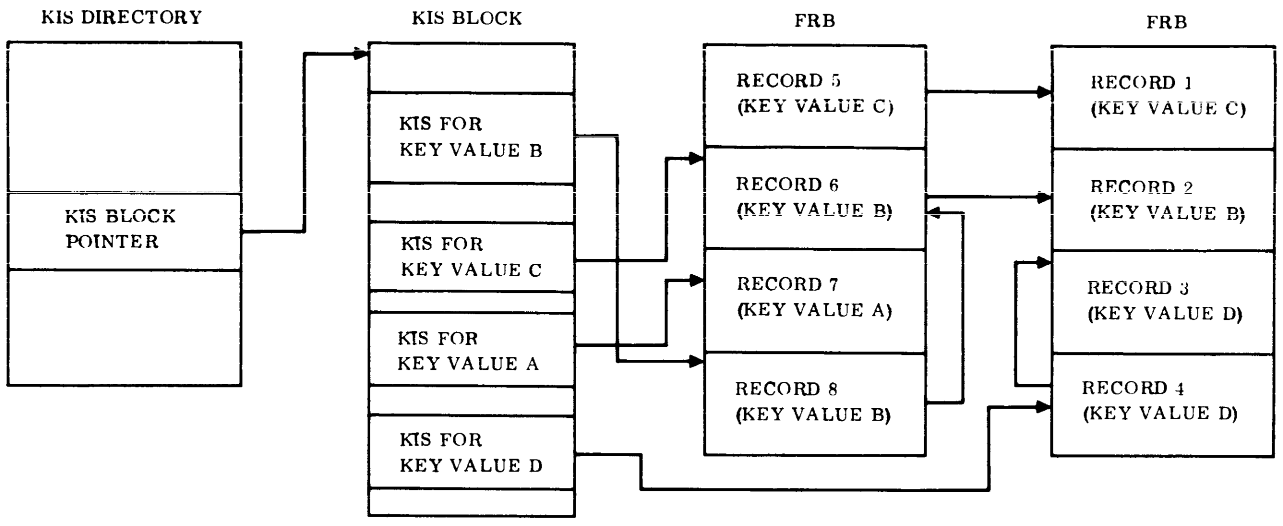


Figure E-2. File Structure for Indexed-Linked File with LIFO-Linking

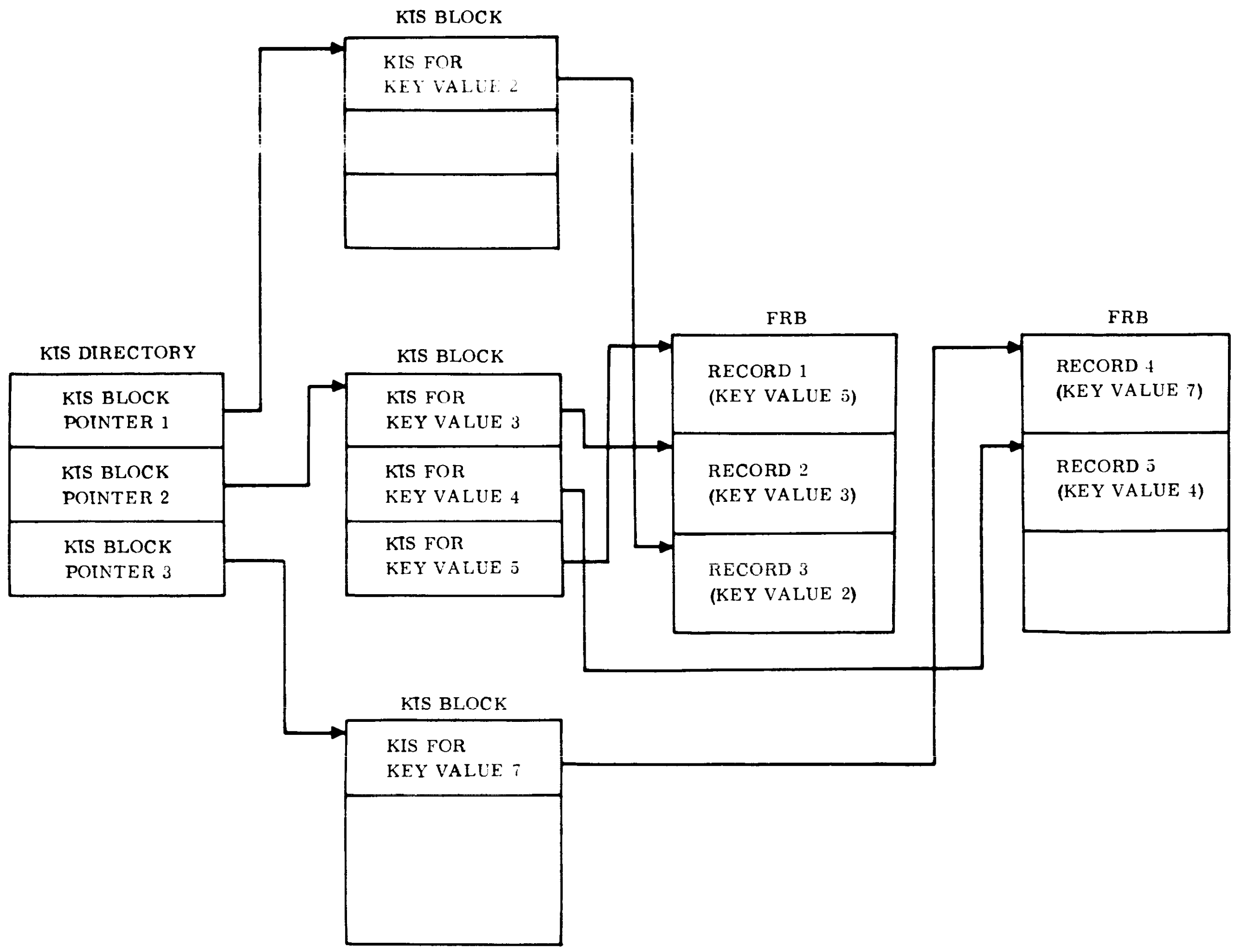


Figure E-3. File Structure for Indexed-Ordered File

FILE MANAGER ERROR MESSAGE F

PRINTING ON COMMENT DEVICE	MEANING	RECOVERY
F.M. ERROR 1	Irrecoverable mass memory error occurred while space was being returned to the space pool. This error may result in invalid space pool threads and/or file space being lost to the File Manager.	The user may autoload and purge all system files, then reload files from a user written backup as described in Section 2.8.

COMMENT SHEET

MANUAL TITLE Control Data® 1700 System MSOS 4 File Manager

Version 1, Software Reference Manual

PUBLICATION NO. 39520600 REVISION A

FROM NAME: ______

BUSINESS
ADDRESS: ______

COMMENTS: This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number.

CUT ALONG LINE

STAPLE

STAPLE

FOLD

FIRST CLASS
PERMIT NO. 333

LA JOLLA, CA.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
SMALL COMPUTER DEVELOPMENT DIVISION
4455 EASTGATE MALL
LA JOLLA, CALIFORNIA 92037

ATTN: PUBLICATIONS DEPARTMENT

CUT ALONG LINE

FOLD

STAPLE

STAPLE

CONTROL DATA
CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN, 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

Litho in U.S.A.