FILE SYSTEM TO SUPPORT TIME SHARING IN A MULTIPROGRAMMING ENVIRONMENT

by

Jerry William Johnson

S.B., University of Houston
(1968)

SUBMITTED IN PARTIAL FULFILLEMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE*

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1970

Signature of Author _____
Department of Electrical Engineering, June 4, 1970

Certified by _____
Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Graduate Students

FILE SYSTEM TO SUPPORT TIME SHARING IN A MULTIPROGRAMMING ENVIRONMENT

by

JERRY WILLIAM JOHNSON

Submitted to the Department of Electrical Engineering on June 4, 1970 in partial fulfillment of the requirements for the degree of Master of Science

ABSTRACT

This thesis presents a design for a flexible, versatile file system to support time sharing in a multiprogramming environment on small core computers. However, many of the concepts and techniques developed in this design may be readily adapted to take advantage of extra core storage available on large core computers. The activities performed by this file system are divided into a hierarchical sequence of logically complete functions. Each logically complete function forms a level in the hierarchical structure of the file system. The overall function of the file system is the hierarchical composition of the logically complete functions performed by each level. This re-entrant file system provides its users with a uniform file structure in the form of virtual memory, a set of hierarchical structured directories, a protected environment which permits and encourages file sharing, and the capabilities to create, open, write, read, link, truncate, delete, and close files with no a priori knowledge of secondary storage devices.

Thesis Supervision: John J. Donovan
Title: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENTS

I express my appreciation to my thesis advisor, Professor John J. Donovan for his active interest, technical assistance, and encouragement; to Project Mac for providing a stimulating environment where many ideas contained in this thesis were developed; to Stuart Madnick and Stephen Zilles for their assistance and enlightening criticism; to the National Science Foundation for financial support; to my wife, Janet, for her patience and perservance while typing my thesis; and also, to my wife, Janet, who has been a constant source of encouragement and understanding through my years of academic study.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER I

INTRODUCTION

Goals, Accomplishments, and Implementation Aids

The initial goal of this thesis was to design a flexible, versatile file system to support a multi-tasking environment on an 8K or 16K IBM 1130 computer, lacking memory protection, a timing clock, and an interrupt-masking facility. Another goal of this thesis was to provide a hierarchical structured file system design which could be easily implemented and debugged.

These goals were accomplished in the following sense. The file system design presented will support multi-tasking and time sharing (for 1130's with timing clocks) environments on the IBM 1130 computer. The file system design provides the users of the system with a flexible, versatile file system which permits and encourages controlled data file sharing. In a larger context, the design approach has been sufficiently machine-independent such that it has considerable applicability to small computers in general. Many of the concepts and techniques presented in this file system design may be readily adapted for use on large computer systems. The file system design has been organized into a hierarchical sequence of logically independent functions or levels,

with well defined interfaces. This internal structure coupled with well defined interfaces provides a conducive environment for independently implementing and debugging each hierarchical level.

The primary contribution of this thesis is a description of a methodology for the design of file systems which provide a versatile set of capabilities simultaneously for several users in a small core environment.

During the implementation phase on the IBM 1130, precautions must be taken to prevent common data bases from being simultaneously modified. This can be accomplished by utilizing a software disable, which must be provided by the multi-tasking monitor, each time a common data base within the file system is being accessed. Time sharing and multi-tasking places the constraint that the file system design must be implemented as reentrant or pure procedures. A re-enterable program must be divided into two logically and physically distinct parts-a constant part and a variable part. The constant part is loaded into memory once and services several tasks concurrently by switching from one task to another task at high speed. This is why the software disable must be used when a particular task is accessing the data bases of the pure procedures.

Hierarchical Modularity of the File System Design

The activities performed by this file system are divided into a hierarchical sequence of logically complete functions. Each logically complete function forms a level in the hierarchical structure of the file system. Each level in the structure can only communicate with one successor ond one predecessor level. In fact, each level in the hierarchical structure can only "call" one predecessor level. The users of the file system interface with the highest level on the hierarchical structure. Each level in the file system hierarchy is more "powerful" than its predecessor level in the sense that it may utilize the capability of its predecessor to perform its logical function.

The rationale for choosing this hierarchical structure is to segregate each logically complete function of the file system in order to make the overall file system easier to design, implement, and verify that it performs its intended function. After the interfaces are well defined, the design of the overall file system reduces to designing a set of independent levels in the hierarchical structure. The overall function of the file system is the hierarchical composition of the logically complete functions performed by each level.

The modular structure incorporated into the design of

this file system is similar to the hierarchical organization presented by S.E. Madnick in a recent M.I.T. thesis (Madnick 69).

Experience with complex file systems has shown that is is a very difficult and time consuming task to localize and identify (debug) the errors in the system during the implementation phase. A goal of this hierarchical design is to present an internally structured file system which provides a conducive environment for efficient "debugging."

Overview of Hierarchical Levels in the File System

The design of this file system consists of five hierarchical levels. These five hierarchical levels are:

1. Device Strategy Module
2. Allocation Strategy Module
3. File Organization Strategy Module
4. Basic File System phase
5. Logical File System phase

The design of these hierarchical levels is presented in "reverse" order since this is the order that they are employed to sequentially process the user's request. The users of the file system interface with the highest level in the hierarchical structure which is the Logical File

System phase. For example, a user's program may call the Logical File System phase to READ file Square Root into core storage from secondary storage. There may be more than one Square Root file on secondary storage. There may be a Square Root file in the System Library, a second Square Root file containing a routine programmed by some user to calculate square roots very fast but only to slide rule accuracy, and a third Square Root file containing a square root routine, perhaps designed by a freshman in Computer Science, which has not been debugged. The function of the Logical File System phase is to transform the symbolic name, Square Root, into a unique file number which identifies the particular Square Root file requested by the user. A more detailed discussion of how the Logical File System phase performs its functions is given in Chapter II. The Logical File System phase now calls the Basic File System phase to read the file corresponding to the unique file number into core.

The Basic File System phase converts the unique file number into the physical description of the file to be read. The physical description of this file provides all the information needed by the next level of the file system to physically locate the file on secondary storage. A more extensive discussion of the functions performed by the Basic File System phase is presented in Chapter III. The

Basic File System calls a File Organization Strategy Module to read the file whose location is specified in the physical descriptor.

The File Organization Strategy Module "knows" that secondary storage is physically divided into distinct records of magnetic memory and that each distinct record has a unique physical address. The FOSM utilizes the information contained in the physical description passed from the Logical File System phase and its knowledge of the organization of files on secondary storage to prepare a list of the physical record addresses which should be read into core. For a meticulous discussion of the FOSM see Chapter IV. The FOSM calls a Device Strategy Module to read this list of physical records into core.

The Device Strategy Module, using a strategy to minimize the overall time, issues commands to the appropriate I/O device to read the physical records requested by the FOSM from secondary storage into core storage. Since the file system has read the requested Square Root file into core storage, it returns control to the user. The detailed discussion of the functions performed by the DSM is presented in Chapter VI.

The Allocation Strategy Module was not discussed in the above example since it is not involved in reading files. When a file is initially written to secondary storage,

the same sequential process through the hierarchical file structure, as presented above, occurs with the Write command replacing the Read command, until the FOSM is called. At this point, the FOSM calls the Allocation Strategy Module to get a set of "empty" physical records on secondary storage for this file. The ASM keeps a "tally" of the addresses of all "empty" physical records on secondary storage. It chooses a set of physical records according to some strategy, updates its "tally" to indicate these physical records are no longer "empty" and returns the addresses of the physical records to the FOSM. The FOSM saves these physical record addresses for future use, prepares a list of physical records and calls the DSM to write the file located in core into these physical records on secondary storage.

CHAPTER II

LOGICAL FILE SYSTEM PHASE

Functional Description

A user's program references each file by a means of a symbolic pathname. It is the function of the LFS phase to convert the symbolic pathname into its unique file identifier. It is the responsibility of the LFS phase to generate and maintain a flexible, versatile, file directory organization for the users of the system. This organization must provide an environment in which file sharing by users may be allowed and controlled. A user should not be allowed to reference the files of other users without their permission.

The description of directory files and the hierarchical file structure provided by the LFS phase will be presented in this section. The concepts of links and keys, along with the features they provide the user, will be discussed. Finally, an outline of the design of the LFS phase will be presented. The detailed logical flowcharts for the algorithms which constitute a specific design for the LFS phase can be found in Appendix A.

Directory Files

There are only two classes of files in the LFS phase. The set of file directories which are maintained by the LFS phase form one class and the set of all other files, referred to as data files for clarity, form the other class. The LFS associates a symbolic name with each of the files in both classes.

A directory is a file of arbitrary length which contains a list of entries. Functionally each entry points to either a data file or to another directory. Specifically, each entry contains five fields. The first field contains the entry name which is identical to the symbolic name of a file directory or data file. An entry name need be unique only within the directory in which it occurs. This condition permits several files to exist simultaneously on secondary storage with identical symbolic names. The second and third fields contain a symbolic volume and index, respectively, that correspond to the unique file identifier assigned to the symbolic name. This unique file identifier is used in resolving the ambiguity associated with files having common symbolic names. The semantics associated with the symbolic volume and index and the rationale for choosing these to represent the unique file identifier will be deferred until the discussion of the LFS phase is undertaken. The fourth

field contains a unique key, if the entry corresponds to a file that is linked to another file; otherwise, this field is empty. The concept of links will be discussed in the following paragraphs. The fifth field contains a set of attributes. The first attribute indicates whether the file associated with this entry is a directory file or data file. The second attribute stipulates whether the file is linked or not linked. Figure 2.1 depicts the format of an entry in a directory file.

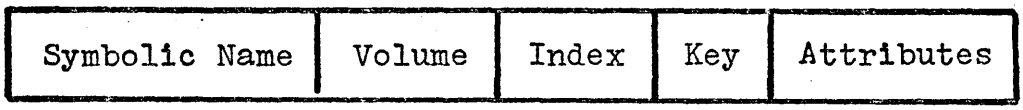


Figure 2.1--Format of Entry in a Directory File

Hierarchical File Structure Without Links

The hierarchical file structure is a set of multi-leveled data structures maintained on secondary storage. Level zero of the file structure contains the base directory, often called the root directory. The unique file identifier of the root directory is known to the LFS. The entries of the root directory contain the unique file identifiers for all directory files and data files which are found at level one of the file structure. There exists one entry in the root directory for each user that has permission to use the computer system. Therefore, if a particular user had more than one file stored on

secondary storage, the unique file identifier corresponding to his entry must point to his file directory located at level one in the file structure. The rationale for having each entry of the root directory correspond to a unique user or class of users is tied very closely to file protection. Since each path through the hierarchical file structure begins in the root directory, a particular user can access only those files in the file structure which are located by a path emmanating from his entry in the root directory. Thus, there is a multi-leveled data structure associated with each user or class of users. A user is said to own all files in the multi-leveled data structure associated with him.

The entries of a user's directory file at level one in the hierarchy contain the symbolic names and unique file identifiers for all of his directory and data files which are found at level two. A user can have as many levels in the hierarchical file structure, independently of the other users, as he desires. In general, any user's directory files at level "i" in the hierarchical file structure contain the symbolic names and unique file identifiers for all of his directory and data files which are found at level "i+1."

Any user's directory file or data file at lever "i" in the hierarchical file structure must correspond to a

unique path through the first "i" levels of the file structure. Therefore, a user may access one of his files at level "i" with a qualified symbolic path name consisting of "i-1" components. The LFS will append the identification of the user to the front of the symbolic pathname specified by the user to get the "i"-components necessary to access the requested file at level "i" in the hierarchical file structure. Hence, the LFS allows a user to access only those files at level "i" of the hierarchical file structure that emanate from his entry in the root directory.

Figure 2.2 illustrates a typical hierarchical file structure consisting of three separate multi-leveled data structures. The numbers 1, 2, 3 correspond to the system's identification of three different users or classes of users. The numbers 1 and 2 point to the directory files on level 1 for user 1 and user 2, respectively.

The files 1, 2, and B, denoted by square symbols, are directory files, and the files X, Y, and Z, denoted in circles, are data files. The directory files and the data files do not necessarily have unique symbolic names. The only requirement is that all file names in each directory be unique. There are three data files in Figure 2.2 named "X"; this ambiguity is resolved by the qualified pathnames "1.B.X," "2.X," and "1.B.B.X." Similarly, the ambiguity associated with the two

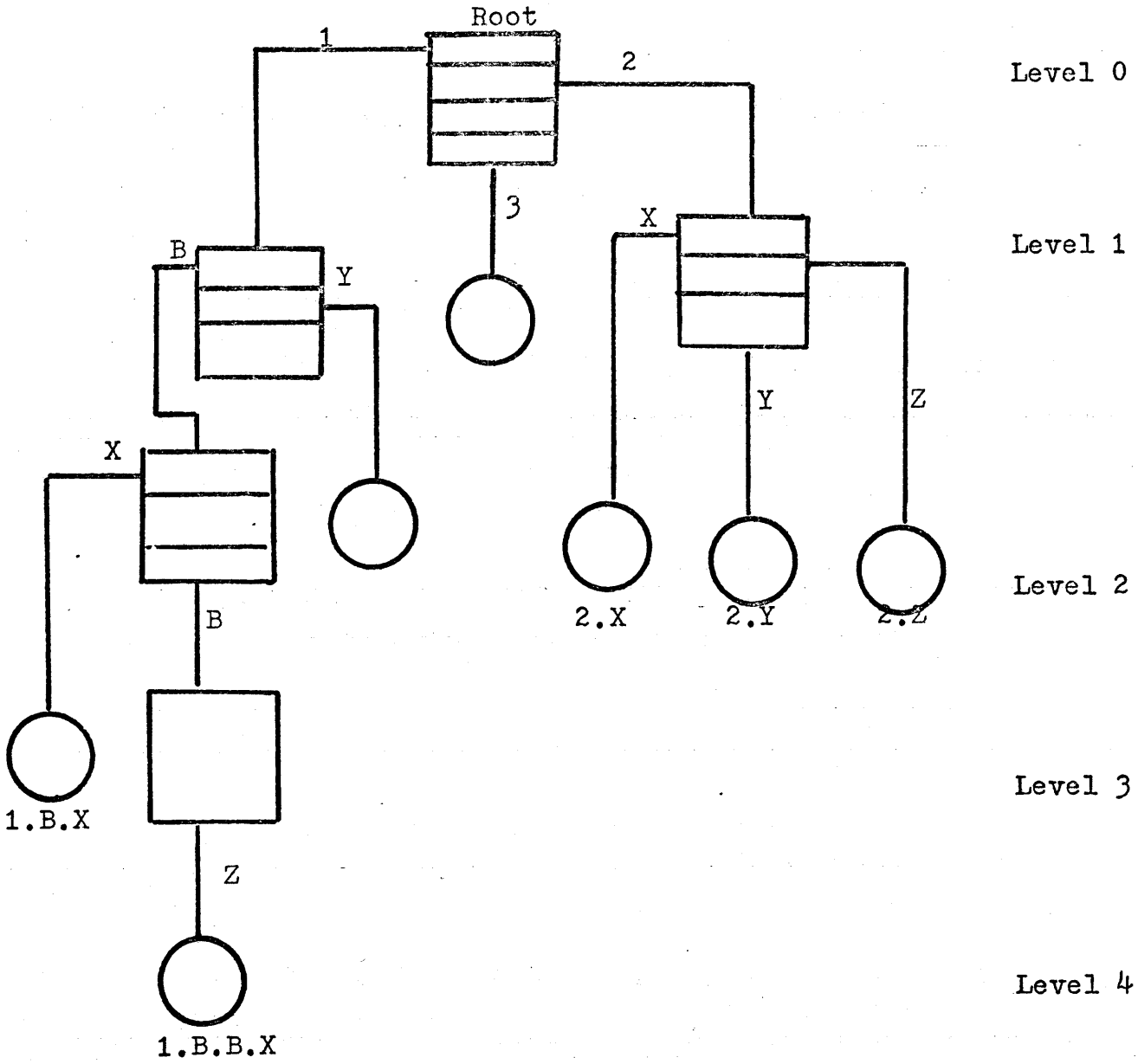


Figure 2.2--Typical Hierarchical File Structure

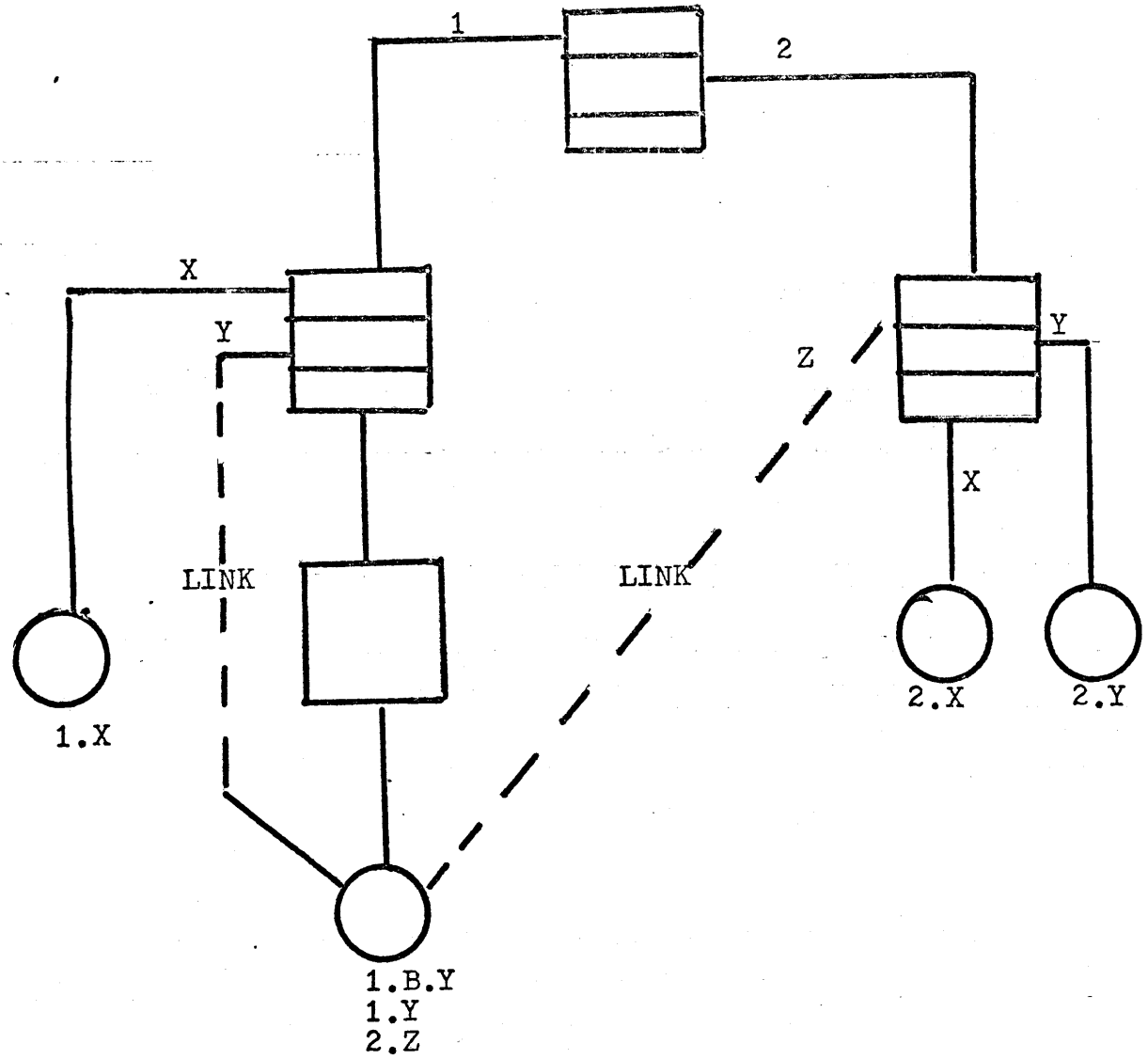


Figure 2.3--Hierarchical File Structure with Links

directory files named "B" is resolved by the pathnames "1.B" and "1.B.B." Note that user 3 has only one data file on level 1 and has no directories. If user 3 had more than one data file, then he would need a directory at level 1 in the file sturcture.

Concept of a Link

Recall that each directory file is owned by a particular user or class of users, and that a user can access only those data files which appear in his directory files. Without links, entries in a user's directory file point to other directory or data files that belong to him in the next level of the file hierarchy. Primarily, links provide a means by which a user can reference data files that do not belong to him from any of his file directories. Secondarily, links allow a user to reference any of his data files from any of his directories.

The following example is given to clarify the concept and flexibility of a link. Suppose user 1 has a data file, "Y," which is located in his hierarchical file structure by the symbolic pathname "1.A.B.Y." Assume user 1 has given user 2 permission to link to file "Y" to read only. Since user 2 can link to file "Y" from any of his file directories for the purpose of reading, suppose he chooses to link to file "Y" from his directory, "C," with

symbolic pathname "2.C." Suppose user 2 decides to rename file "Y" as file "Y2." Now the link is accomplished by making an entry "Y2" in file directory "C" such that the unique file identifier of "Y2" is the unique file identifier of "Y." When user 2 accesses his data file corresponding to "2.C.Y2," the LFS phase will map this symbolic pathname into the unique file identifier corresponding to file "1.A.B.Y." Data file "2.C.Y2" is said to be linked to data file "1.A.B.Y."

Hierarchical File Strucutre with Links

The hierarchical file structure presented protects a user's set of files since no other user with a different system identification can access those files. If a user is successful in masquerading as another user (convincing the computer system that he is another user), then this protection schema fails. This hierarchical file structure does not allow file sharing and controlled access among individual users and classes of users.

The flexibility and versatility of the above hierarchical file structure can be enhanced by allowing links to data files to be superimposed on this structure. File sharing and controlled access among users is permitted and supervised through the use of links. The allowable links to a particular file are controlled by

the owner of that file. The means by which permission is given to link to a file is deferred until the discussion of the BFS phase is undertaken. The extent to which links modify the hierarchical file structure will be discussed here.

Links modify the basic hierarchical file structure by allowing data files to be referenced directly from any directory in the file hierarchy. Figure 2.3 shows a hierarchical file structure with links. User 2 has linked to data file "Y" which belongs to user 1. This link allows the LFS to access data file "Y" for user 2 with the qualified pathname "2.Z." Note that user 2 can access this file by simply specifying the symbolic name "Z." This is the mechanism by which the links allow users to share data files. The extent of sharing is controlled by the owner of a data file since the owner gives permission to link and gives the conditions of a link. For example, a link might be allowed with the condition of read only to some users and write only to other users. Links also help to eliminate the need for duplicate copies of sharable files. The other link shown in Figure 2.3 serves as a shortcut to a data file located somewhere else in the hierarchy associated with user 1. Thus, user 1 can reference file "Y" with either of the two pathnames "Y" or "B.Y."

Since links are not allowed to data files that have not been created, the links are always superimposed on the existing hierarchical file structure. Loops can never occur in the hierarchical file structure.

Concept of Keys

Functionally, keys are used to confirm the physical existence of linked data files. It is necessary to confirm the existence of linked data files because of a condition which may occur when some data file "Y" is linked to some data file "Z" and data file "Z" is subsequently deleted. Since the unique file identifier that was assigned to data file "Z" has been deleted, it may at some later time be assigned to a new data file "X." The critical condition occurs when the linked data file "Y" is accessed after data file "X" has been created. The symbolic name of data file "Y" is mapped into its unique file identifier which now corresponds to the unique file identifier of "X." The result may be disasterous since data file "Y" is now actually linked to data file "X" in lieu of data file "Z." Keys provide a means by which the LFS and BFS phases, working in conjunction, can solve this conflict.

Before presenting exactly how the keys permit the above conflict to be resolved, the relevant interaction

between the LFS phase and the BFS phase is outlined.

When a file is created, a unique key is generated by the LFS and entered into the directory entry along with the symbolic name and unique file identifier. The BFS phase uses the unique file identifier to access the file descriptor associated with the created file and inserts the key into the file descriptor. The directory entry of a linked file is assigned the key and unique file identifier of the data file to which it is linked. Therefore, no new file descriptor is created for the linked file, since file descriptors have a one-to-one correspondence with the unique file identifiers. The essential point is that the key associated with a file is stored in the file directory and in the file descriptor.

The means by which keys solve the above conflict is now presented. Since no two keys are identical, the descriptor of data file "X" will contain a different key from the key contained in the directory entry of data file "Y." The BFS using the unique file identifier of file "Y" will still access the new descriptor associated with data file "X." However, the key from the directory entry of file "Y" will not agree with the key in the descriptor of file "X." Thus, the conflict is resolved since the BFS detects that the file to which "Y" was originally linked has been deleted.

Outline of Design of the Logical File System Phase

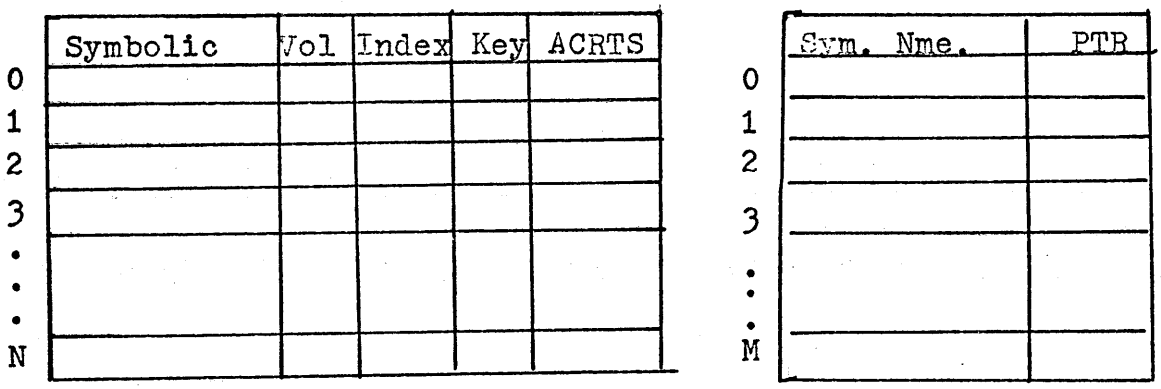
The Logical File System consists of a mainline module, LFS, that calls one of a set of submodules. Each submodule corresponds to one of the ten user commands processed by the Logical File System phase. The overall tasks of the Logical File System, along with those submodules which perform that task, are:

CREAT	1.	Creates a directory file or data file for a user in his hierarchical file structure.
OPEN	2.	Opens a file for accessing by transferring the directory entry of this file from secondary storage into a core resident table (the Active File Directory).
CLOSE	3.	Closes a file by deleting its directory entry from the Active File Directory.
READ	4.	Maps the symbolic pathname of the referenced file into a unique file identifier and key by use of the Active File Directory if the command is allowed.
WRITE	5.	Same submodule as 4.
TRCAT	6.	Same function as 4.
PROT	7.	Same function as 4.

DELET	8.	Deletes a file for a user in his hierarchical file structure.
LINK	9.	Creates a data file for a user in his hierarchical file structure and links it to the specified data file if the link is allowed.
ULINK	10.	Same as 8.

Data Bases of LFS

The Active File Directory and the Free List are the data bases which belong to the LFS phase. The LFS also utilizes a system wide buffer in its search through the file directories for requested files. The structure of the AFD and the Free List are shown in Figure 2.4.



a. Structure of AFD

b. Structure of Free List

Figure 2.4--a. Structure of AFD; b. Structure of Free List

Since the size and number of file directories will generally prohibit them from being maintained in core storage, they are stored on secondary storage devices. The AFD permits a set of N file directory entries to be core resident for efficient access. Since the access times of secondary storage devices are inherently much slower than core storage access times, the time efficiency of requests to the file system would be drastically reduced if each request had associated with it a sequence of secondary storage accesses to locate the unique file identifier. The LFS phase utilizes the AFD coupled with the Free List to efficiently perform its frequent function of mapping a symbolic pathname into a unique file identifier (volume and index).

To make an entry into the AFD, the OPEN submodule searches through the hierarchical file directories until it finds the entry corresponding to the file to be opened. This directory entry is transferred to the AFD and the condition associated with the OPEN command is put into the flag field of the AFD. The remaining components of the symbolic pathname are chained together in reverse order in the Free List. The last $\log_2 M$ bits of the ACRTS field serves as an index which connects the entry of the AFD to the chain in the Free List.

Algorithms of the LFS

The detailed logical flowcharts of each of the submodules making up the LFS phase are given in Appendix A. Flowcharts for two utility routines, FDMGT and ACESS, which search the file directory hierarchy and perform the actual accessing, deleting, and inserting of entries into directories, are also given in Appendix A. The syntax and semantics of the arguments required by each of these submodules are found with the flowcharts. The error handling functions performed by the submodules are incorporated into the flowcharts.

Protection Performed by the LFS

The LFS phase checks each access to confirm that it checks the condition for which the file was opened. This phase prevents any user from creating two or more files in the same directory with identical symbolic names. The organization of the file hierarchy prevents any user from accessing files of another user unless links have been made to those files.

CHAPTER III

BASIC FILE SYSTEM PHASE

Functional Description

The Basic File System phase must convert the unique file identifier from the Logical File System phase into a file descriptor. The file descriptor provides all information needed by the next level of the file system, the File Organization Strategy Module, to physically locate the file. The file descriptor provides a means by which the access rights of a file can be dynamically modified by the owner of a file. The BFS phase decides if a file may be opened by verifying access rights and checking read/write interlocks.

File Descriptor

There must be a single file descriptor for each file that resides on secondary storage regardless of how many symbolic names the file may have or of how many different file directories in which it may be found. There are two reasons why the file descriptor is not included in the file directory along with the symbolic name. First, it is more efficient in time and space to maintain one copy of the file descriptor instead of a copy for each symbolic file in the file hierarchy which

eventually points to the same physical file. For example, suppose an owner of a file decided to change the access rights of one of his data files from read and write to read only, then the descriptor of every file in the file hierarchy which was linked to this file would have to be found and changed. Second, it is much easier to insure a unique mapping between the symbolic file and the corresponding physical file through a single file descriptor in a sophisticated environment which allows a single file to be referenced by different names and which permits links that allow a file to be referenced from various directories in the file hierarchy or from different users.

Since each file requires a file descriptor, the set of file descriptors will reside on secondary storage. The file descriptor for a file must reside on the same volume as the file. This is a reasonable condition since, if the volume is mounted, then the file descriptor and the file can both be accessed. A volume refers to the physical medium on which the information is stored where a device refers to the I/O mechanism used to read or write information. For most drums and many disk units, the device and volume are inseparable. However, for tape units and many of the smaller disk units, the volume, magnetic tape reel, and disk pack, respectively, are removable.

For each volume, there is a Volume Descriptor File (VDF) whose fixed length entries contain the file descriptors for each file stored on that volume. Since the entries have a fixed length, the location of a particular file descriptor is specified by its index within the VDF. The descriptor of the VDF is the first descriptor in the VDF. The rationale for selecting a symbolic volume and index to represent the unique file identifier was to allow the file descriptors to be accessed without having to search the VDF. The symbolic volume specifies the volume containing the VDF, while the index specifies the position within the VDF which contains the file descriptor. The information which defines the physical location of the VDF is contained within its descriptor.

To facilitate efficient access to the file descriptors, an Active Volume Descriptor File (AVDF) is maintained in core storage for all the active or open files. The AVDF is the only core resident data base of the BFS phase. Each of the fixed length entries in the AVDF can contain one file descriptor. It is feasible to have a core resident AVDF, since the number of active file descriptors is in general only a small fraction of the total number of file descriptors. The file descriptors of the VDF'S are always resident in the AVDF since one of them is used in the access of each file descriptor. The file descriptor entries in

the AVDF have the format shown in Figure 3.1. The functions of each of the fields contained in the file descriptor are discussed in the following sections.

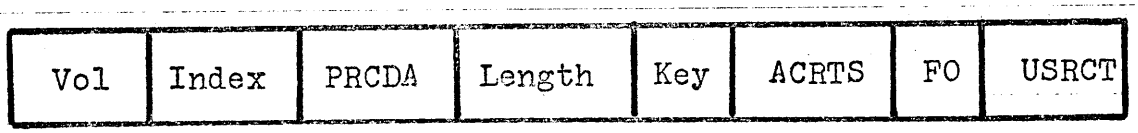


Figure 3.1--Format of a File Descriptor Entry in AVDF

Controlled Access Rights

When a user creates a file, he specifies the initial access rights (read only, write only, etc) associated with his file. He has the capability to modify these access rights at a later time. These access rights are preserved within the ACRTS field of the file descriptor. Controlled data file sharing is tied very closely to the access rights contained in the file descriptor. A simple means by which a versatile set of access rights are allowed and controlled is given in the Outline of Design of the Basic File System Phase. The BFS phase decides if a file may be opened to be accessed in a particular way. It also confirms the physical existence of a data file to be opened by comparing the key passed as an argument in the OPEN command from the LFS phase with the key in the file descriptor. If the keys fail to match, the data file linked to has been deleted. Once a file has been successfully opened, it is the responsibility of the LFS

phase to confirm that a file is really being accessed according to the permission granted in the OPEN request.

Read/Write Interlocks

A data file is allowed to be open for the sole purpose of reading by any number of users. Only one user at a time is permitted to open a file for writing or for reading and writing. The BFS phase enforces these read/write interlocks. The count of the number of users who simultaneously have a file open for reading is saved in the USRCT field of the file descriptor in the AVDF. The value of the user count is used by the BFS phase to prevent it from closing a file prematurely when a set of users are reading a data file concurrently.

Outline of Design of the Basic File System Phase

The Basic File System consists of a mainline module, BFS, that calls one of a set of submodules. Each submodule corresponds to one of the nine commands processed by the BFS phase. The overall tasks of the Basic File System, along with those submodules which perform that task, are:

CREAT 1. Creates a file descriptor entry in the VDF for a file.

OPEN 2. Opens a file for accessing by transferring the file descriptor of this

file from the VDF on disk to the core resident AVDF if the OPEN command is allowed.

CLOSE	3.	Closes a file by deleting its file descriptor from the AVDF.
READ	4.	Retrieves the information from the AVDF which is used by the File Organization Strategy Module to physically locate the file.
WRITE	5.	Same submodule as 4.
TRCAT	6.	Same function as 4.
PROT	7.	Modifies the access rights of the file descriptor in the VDF as specified by the owner of the file.
DELET	8.	Deletes a file descriptor from the VDF maintained on secondary storage.
LINK	9.	Checks the file descriptor of the file to which a link is requested to see if the link is allowed.

Data Bases of BFS

The BFS phase generates and maintains a set of VDF's, one on each volume. Each VDF is actually a directory containing an ordered sequence of fixed length entries. Each entry represents a file descriptor for some file on

the same volume. The format of each entry of a VDF is given in Figure 3.2.

PRCDA	Length	Key	ACRTS	FO

Figure 3.2--Format of Entry in VDF

The structure of the Active Volume Descriptor File (AVDF) is presented in Figure 3.3. The core resident AVDF contains a file descriptor for each OPEN file. The OPEN submodule transfers entries into the AVDF and the CLOSE submodule removes these entries.

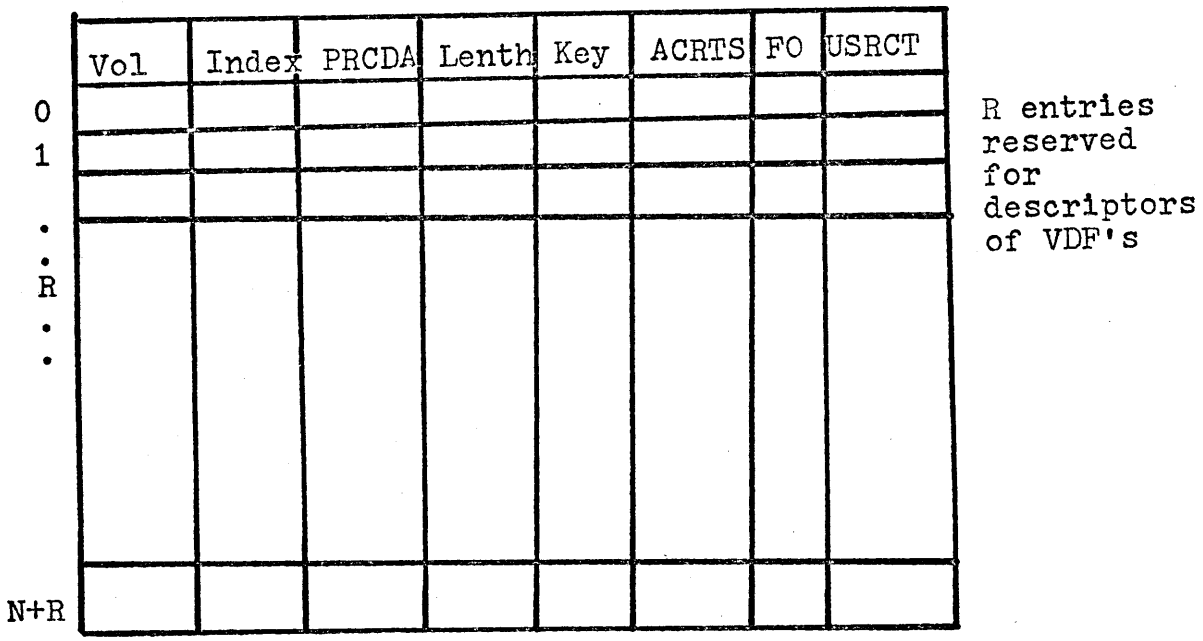


Figure 3.3--Structure of Active Volume Descriptor File (AVDF)

The access rights field of a file descriptor is divided into a set of subfields as depicted in Figure 3.4.

Figure 3.4--Subfields of ACRTS Field

Subfield A is used to determine if a link can be made to the file. Subfield B is used to specify the access rights of an allowed Link. Subfield B contains one of the access rights, Read, Write, Read/Write, or Protected. Subfield C contains either Read, Write, or Read/Write and is used to specify the access rights of the owner of the file. The owner of a file is the only user authorized to delete his files.

This organization allows certain classes of global control of data file sharing to be easily implemented. For example, suppose Subfield A allows a Link; Subfield B stipulates Read, and Subfield C permits Read/Write. Then, the owner of the file is authorized to make Read/Write accesses, and Links to the file, for the sole purpose of Reading, are allowed to every user of the file system. In this case, as well as all other cases, except when Subfield B is protected, the verification of access rights can be made directly from the information contained in the file descriptor.

For the case that occurs when Subfield B is Protected,

verification of the access rights of the LINK command or the access rights of an OPEN command when the file to be opened is linked is made indirectly from the information contained within the file descriptor. The extra processing overhead for Protected data files due to the associated indirection is accrued by only the above two commands. A Protected data file means that access rights are assigned to users on an individual basis at the descretion of the owner. For each Protected data file, the BFS phase maintains an Access Rights File (ARF) which contains the identification of each user given permission to link and the access rights permitted with that link. If a data file is Protected, the file descriptor of the data file in the VDF is immediately followed by the file descriptor of its ARF. Therefore, the symbolic volume and index fields of the file descriptor in the AVDF for the Protected data file can be used to access the file descriptor of the ARF. One needs simply to use the symbolic volume and add one to the index to generate the unique file identifier of the ARF. After accessing the file descriptor of the ARF, the ARF can be read into a buffer and searched for the identification and access rights of the user requesting use of the data file.

The decision to Protect a data file must be made when the file is created so that adjacent pairs of file

descriptors can be assigned.

If data file sharing on a global basis instead of an individual basis is satisfactory for a particular implementation of the file system design, then the design of the LINK and OPEN submodules could be simplified by prohibiting the protect feature.

The physical record address (PRCDA) field of the file descriptor specifies the address of the physical record which contains the mapping function used by the File Organization Strategy Module to map logical file requests into physical file requests. The volume containing this physical record is given within the symbolic volume field.

The file organization (FO) field is used to determine the organization of the file on secondary storage. The appropriate File Organization Strategy Module at the next level in the file system is in general selected according to the file organization contained in the file descriptor. Since only one File Organization Strategy Module is presented in this design, the FO field is not actually required. It was incorporated into the file descriptor to allow other File Organization Strategy Modules to be easily added if they were needed and if sufficient core memory was available.

Algorithms of BFS

The detailed logical flowcharts for the algorithms of the submodules contained in the BFS phase are given in Appendix B. The arguments required by each of these submodules are found with each flowchart. The error handling functions performed by the submodules are incorporated into the flowcharts.

Protection Performed by the BFS

The BFS phase confirms the physical existence of linked files by verifying that the key in the OPEN command is the same as the key in the file descriptor. The BFS will not allow files to be opened or linked unless the accessing request stipulated in the OPEN and LINK commands agrees with the permitted accessing rights contained in the file descriptor.

CHAPTER IV

FILE ORGANIZATION STRATEGY MODULE

Functional Description

The File Organization Strategy Module is responsible for the physical organization of a file on secondary storage volumes. The primary function of the File Organization Strategy Module (FOSM) is to transform each request to transfer a portion of a file between core memory and the file's "virtual" memory into a collection of requests which can be used to transfer the same portion of the file between core memory and physical or "buffered" secondary storage memory. Stated more simply, the FOSM maps logical file addresses into physical record addresses. "Buffered" secondary storage memory refers to the physical records of secondary storage which are currently core resident in the I/O Buffer Management System. In order to minimize I/O requests to secondary storage devices, physical records contained in buffers are transferred directly to the stipulated core areas. In the ensuing design, the FOSM is delegated the duty to detect and transfer buffered physical records. It is the responsibility of the FOSM to interact with the Allocation Strategy Module to dynamically allocate and deallocate physical records for files as required.

The FOSM selects a volume on a secondary storage device when a file is created unless a symbolic volume is specified by the creator. The FOSM maps symbolic volume names into physical volume addresses.

Physical Records and Volumes

A physical record corresponds to a unit of transmission between core storage and a volume through an I/O device. A physical record is generally capable of containing several computer words. For example, on the disk volumes of the IBM 1130, a physical record consists of 320 computer words. Each physical record on a volume has a unique address by which it may be accessed.

A volume is a single unit of secondary or external storage, all of which can be read or written by a single access mechanism called an I/O device. A volume is usually an entire disk, tape, or drum and may be dismountable. The disk volumes on the disk drives provided by the IBM 1130 are dismountable.

Virtual File Memory

Each file is an ordered sequence of addressable elements. The size of each addressable element is the same size as an addressable element in core storage. Thus, each file has the form of a "virtual" core memory.

The real purpose of the file system is to provide an easy, reliable means by which a specified number of elements may be transferred between "real" core memory and the "virtual" memory of the file system. A file's virtual memory may be much larger than real memory. In fact, each file is allowed to be arbitrarily long. When a user wants to read or write a portion of a file, he specifies the particular portion in the file's virtual memory. For example, a user may request the file system to read 600 words, starting at address 2,000 within file Beta, into core starting at location 12,250. It is the function of the FOSM to transform this logically contiguous virtual memory 600-word portion into its physical location on secondary storage. In general, the physical location, corresponding to a contiguous area in a file's virtual memory, is designated by a set of physical records and/or parts of physical records.

Several objectives of the file system are realized through the technique of virtual file memory. The virtual file memory provides the user with a flexible and versatile uniform file structure. The virtual file memory provides a shield between the user and the "obscure" mechanisms required by the file system to successfully interact with the secondary storage device. Thus, the user need have no a priori knowledge of the pecularities or the physical

organization of the secondary storage devices to triumphantly use the file system. Since only the virtual memory of a file is addressable by a user, the file system has complete control over the dynamic and automatic allocation of secondary storage for all files. The schematic relationship between core storage, virtual file memory, and physical records on secondary storage is exhibited in Figure 4.1.

Indexed File Organization Strategy

An indexed file organization strategy is the scheme incorporated into the design of the FOSM. In this strategy, the physical record addresses are kept in a table or file which is "indexed" by use of logical record numbers. The FOSM generates the logical record numbers by partitioning the file's virtual memory into an ordered set of logical records having the same size as a physical record. Logical record 1 of virtual memory has the physical record address found in entry 1 of the table; logical record 2 has the physical record address found in entry 2, etc. The means by which the FOSM organizes its data bases to convert logical records into physical records while attempting to minimize the I/O operation required to update the data bases is present in the Design of the FOSM section.

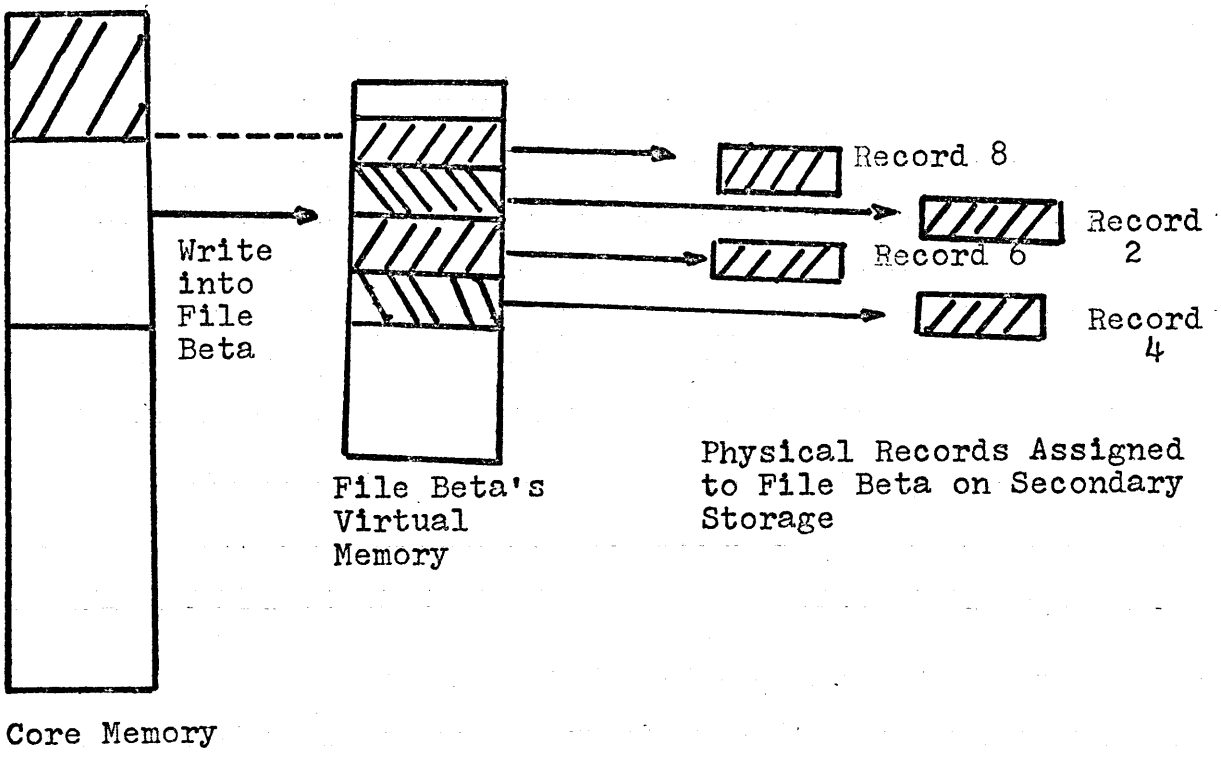


Figure 4.1--Schematic of Relationship between Core Memory, Virtual File Memory, and Physical Records

The rationale for choosing this particular strategy over other common strategies such as sequential file organization and linked file organization is accredited in part to the flexibility and generality of this strategy and in part to the environment of the IBM 1130. The direct access secondary storage devices of the IBM 1130 configuration provide excellent conditions for random or direct access files which are particularily adaptable to a multi-tasking environment.

Of the above strategies, the indexed file organization strategy is the only one which allows efficient direct access files. The core memory available on a particular IBM 1130 for the resident file system may be limited to the extent that room for only one FOSM is permitted. The capabilities provided by the indexed file organization strategy are more adaptable and versatile than the capabilities of either of the other strategies.

The indexed file strategy may simulate the sequential file scheme, using a sequential allocation module, in the sense that sequential physical records are assigned to a file. The argument generally given for assigning consecutive physical records to a file is to minimize device latency and access time. However, in a multi-tasking environment, a common condition is to have more than one file actively in use on the same device. This

produces a state in which the read/write mechanism is switching rapidly among many active files. This condition does not give overwhelming credence to the argument that sequential files minimize latency and access times in a multi-tasking environment.

When space becomes available on a secondary storage volume due to files being deleted and truncated, it usually appears in disjoint fragments throughout the volume. The indexed file organization allows the logical records of the file's virtual memory to be "scattered" over the random-accessed storage volume. This technique permits files to dynamically expand and contract.

Finally, the indexed file organization strategy provides an efficient means to allow "sparse" files such as files containing "hash coded" and random entry tables. A "sparse" file may be characterized as having a much larger virtual memory than the physical memory actually assigned to the file on secondary storage. For pedagodical reasons, assume a user created a file and transferred data into only the portions of the virtual file's memory corresponding to logical records 1, 50, and 100. At this point in time, the length of the file's virtual memory is one hundred times the number of elements in a physical record. However, only four physical records are required to represent this file. One physical record

is needed to contain the physical record addresses and three physical records are needed to contain logical records 1, 50, and 100. Additional physical records will be dynamically allocated as required for a sparse file as more information is written into the file.

A request to read an "unwritten" portion of a sparse file may occur. The FOSM realizes this, while processing the Read request, when it detects that physical record addresses have not been assigned to the requested portion of the sparse file. In such a case, the FOSM returns zeros to give the illusion that the unspecified contents of a sparse file are initialized to zero.

Design of a File Organization Strategy Module

The indexed File Organization Strategy Module (FOSM) is a mainline module that calls one of a set of submodules. Each submodule corresponds to one of the five commands processed by the FOSM. The overall tasks of the FOSM, along with those modules which perform that task, are:

CREAT

1. Chooses a volume for a file that is being created unless the creator of the file has specified a symbolic volume name, verifies that a specified volume is mounted, and calls ASM to allocate a physical record for the

File Index Table.

READ 2. Maps virtual file memory requests into physical record requests and transfers physical records in buffers immediately to the user.

WRITE 3. Maps virtual file memory requests into physical record requests, allocated physical records to files as needed, and transfers physical records in buffers immediately to the user.

CLOSE 4. Updates disk copy of modified core resident data bases associated with specified files including modified physical records in buffers.

TCATE 5. Reduces the length of a file by calling ASM to deallocate physical records assigned to truncated portions of files.

Mapping Virtual File Memory into Logical Records

The virtual file memory is logically partitioned into an ordered set of logical records. The size of a logical record is equal to the size of a physical record of this file. Figure 4.2 displays the concept of partitioning a virtual file memory into logical records.

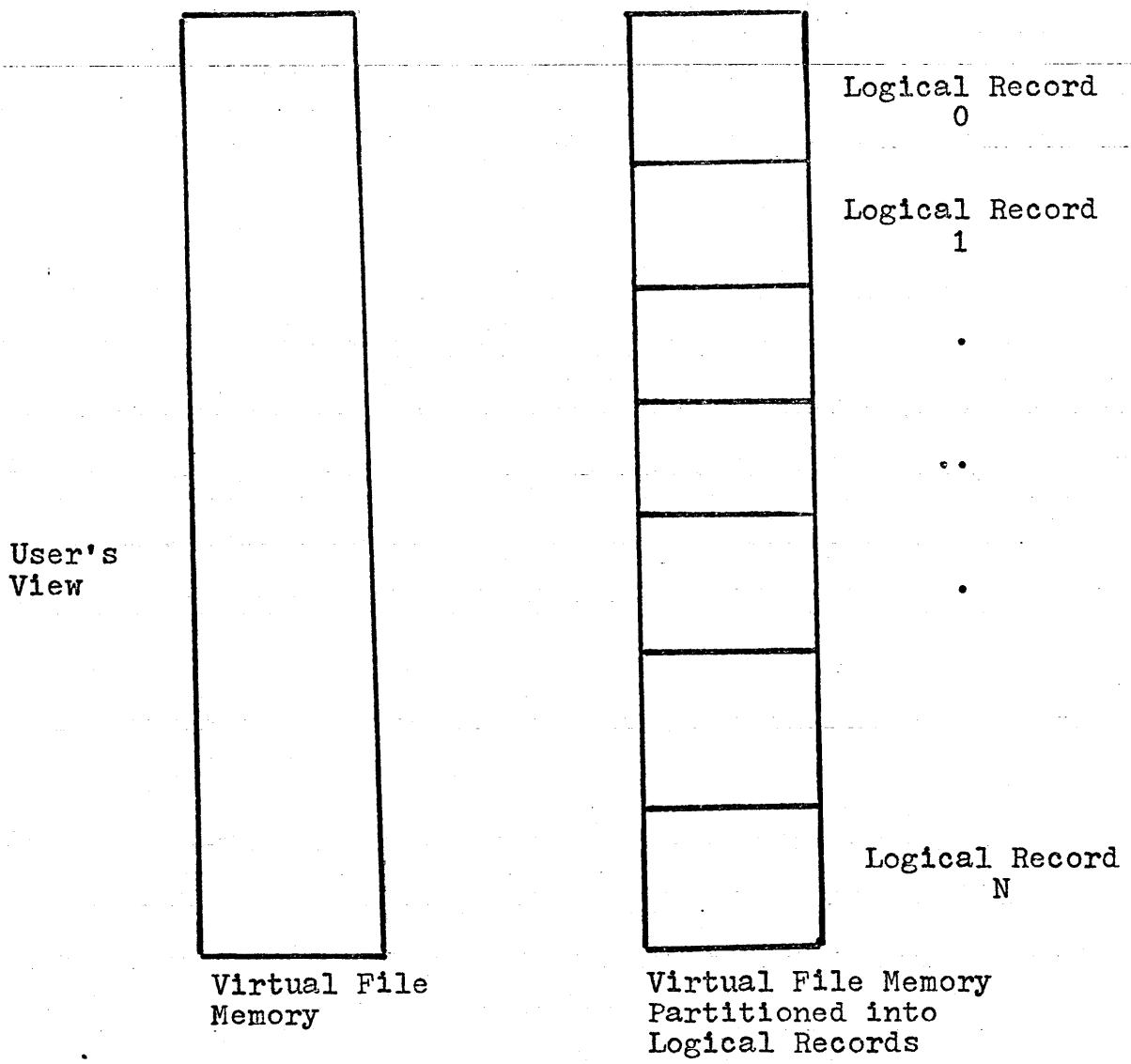


Figure 4.2--Scheme of Partitioning a File's Virtual Memory into Logical Records

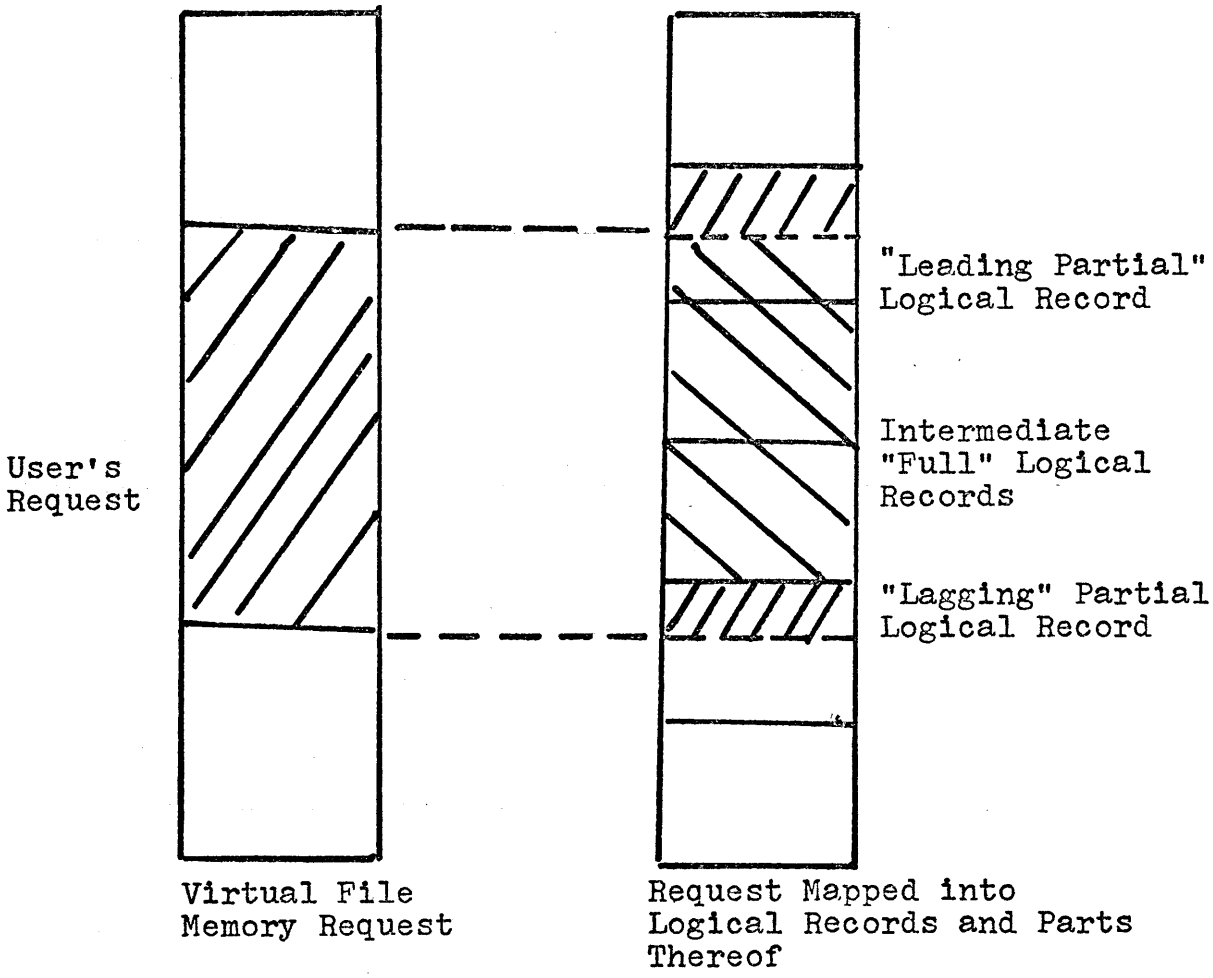


Figure 4.3--Virtual File Memory Request Mapped into Logical Records and Parts Thereof

The virtual file memory is partitioned into logical records in order for the FOSM to associate distinct physical records with each logical record. When a request is made to the FOSM to read or write a portion of the file's virtual memory, the Prepare Logical Record List (PLRL) routine is called which maps the request into the appropriate collection of logical records and parts thereof. This mapping involves only simple mathematical manipulations of the file address and core address specified by the user in his read or write request. A requested portion of virtual file memory and the corresponding collection of logical blocks is exhibited in Figure 4.3 found on the preceeding page.

The output of the PLRDL routine is called the Logical Record List. The format of the Logical Record List is shown in Figure 4.4

	LRCDI	CA	Index	Num	CT
"Leading" Entry					
"Intermediate" Entry					
"Lagging" Entry					

Figure 4.4--Format of Logical Record List

The format of the Logical Record List simplifies the tasks of mapping logical records into physical records and controlling the transfer of partial physical records

through buffers.

The three entries represent the information which describes the leading partial logical record, the set of intermediate full logical records, and the lagging partial records, respectively.

The Logical Record Index (LRCDI) field of an entry contains the index of the logical record described by that entry. The Core Address (CA) field stipulates the beginning of the user's area in core storage corresponding to the logical record of that entry. These two fields pertain to the first intermediate full logical record of the second entry. The Index field specifies the index within the logical record corresponding to the first record of the logical record contained in the read/write request. The Number (Num) field gives the number of words within the logical record contained in the read/write request. The Count (CT) field contains the number of logical records associated with each entry. The CT field of the record entry specifies the number of intermediate full logical records. The CT field of any entry is zero when that entry is not required in a particular read/write request.

File Index Table

A File Index Table (FIT) contains the mapping function

used by the FOSM to map logical records into physical records. There is a FIT maintained on secondary storage for each file in the file system except the FIT. The FIT is actually a chained file having the structure shown in Figure 4.5.

Each entry of the FIT is indexed by a logical record number. Entry 0 corresponds to logical record 0; entry 1 corresponds to logical recore 1, etc. Each entry in the FIT contains the physical record address of the logical record defined by the index of the entry if the physical record has been allocated; otherwise, the entry contains a zero to indicate that a physical record has not been allocated. The FOSM is responsible for the generation and deletion of entries in the FIT as a file "grows" and "shrinks." This indexing scheme incorporated into the FIT permitts logical records to be mapped into physical records without having to search the FIT.

The last entry of each physical record assigned to the FIT contains the physical record address of the next physical record assigned to the FIT. The last physical record in the chain is denoted by a zero in its last entry. Additional physical records are dynamically assigned to the FIT as needed by the FOSM interacting with the Allocation Strategy Module.

For most files only one physical record is required

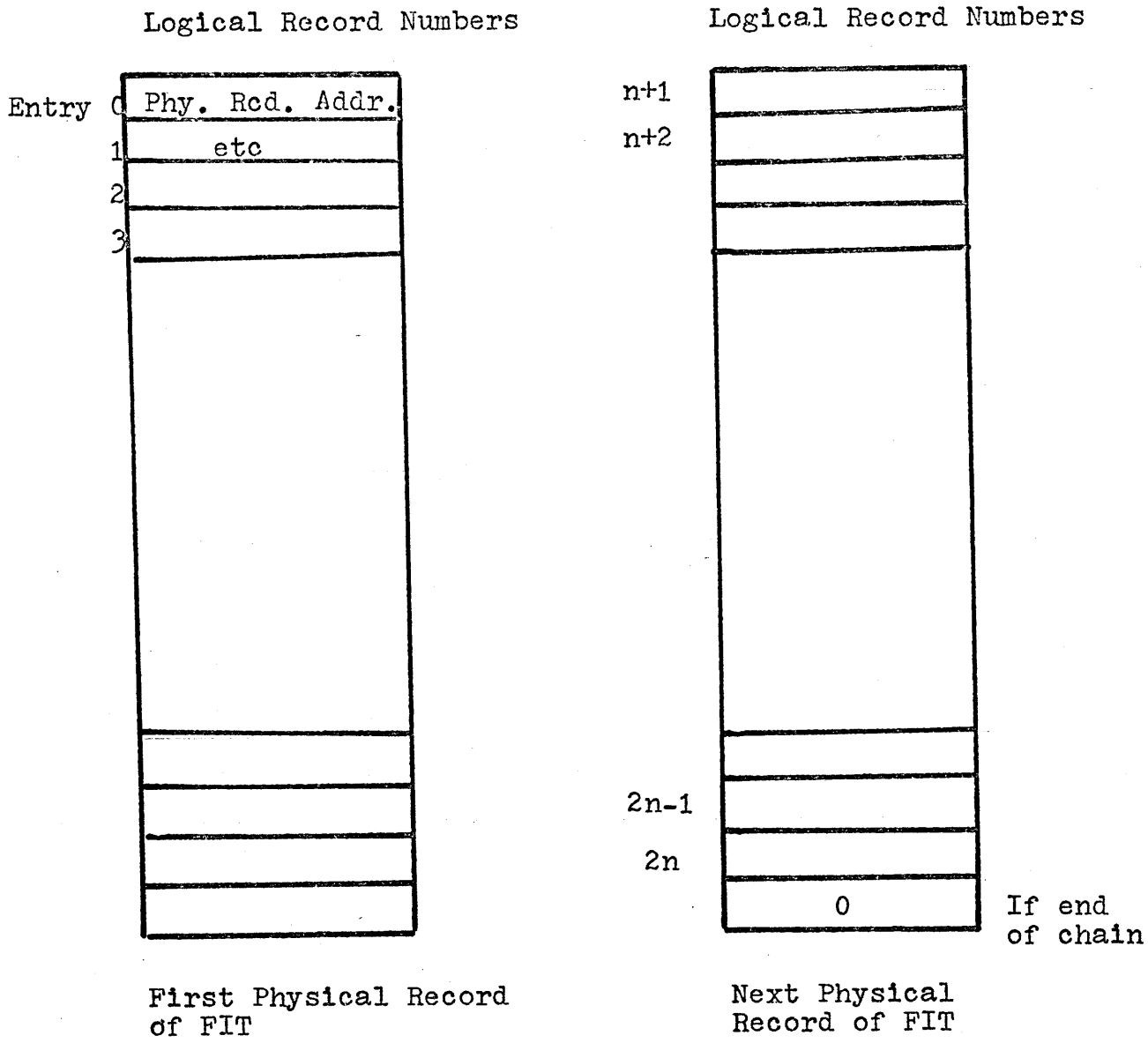


Figure 4.5--Structure of the File Index Table

for the FIT. For example, in the IBM 1130 environment with 320 word physical records, a single physical record of the FIT may contain the physical addresses of 319 physical records which represents the mapping function of a file 102,080 words long.

The physical record address of the appropriate FIT is one of the arguments contained in each Read or Write command issued to the FOSM by the BFS phase.

In order to better understand how the BFS phase is able to specify the physical record address of the FIT the following brief digression is given. At the time a file is created, the FOSM assigns and initializes a physical record for the FIT on the volume which will contain the file. The address of this physical record is returned to the BFS phase. As the BFS phase creates the file desceiptor, it enters the address of the physical record for the FIT into the physical record address (PRCDA) field of the file descriptor. When a user makes a request to Read or Write a file, the BFS phase employs the unique file identifier to locate the file descriptor in the core resident Active Volume Descriptor File. The physical record address of the FIT is extracted from the file descriptor and the FOSM is called.

Active File Index Table

Certain sections of the File Index Table must be in core in order to map logical records into physical records. Since sufficient core storage is not available on the IBM 1130 system to keep the File Index Tables for all opened files in core, contiguous "sections" of the File Index Tables are maintained in core, for the active or opened files, in an Active File Index Table (AFIT).

The Active File Index Table should be structured to allow efficient mapping of logical records into physical records. The contiguous sections of the File Index Tables contained in the Active File Index Table should be "large" to minimize the number of I/O operations required to "shuttle" sections back and forth between the File Index Tables maintained on secondary storage. However, the sections contained in the Active File Index Table are required to be "small" in order to conserve core storage. Thus, a compromise must be made on the lenght of the sections allowed in the Active File Index Table. The structure of the Active File Index Table is delineated in Figure 4.6.

The Active File Index Table is divided into two logical parts. The first part is an index to the second part which contains the "active" sections of the File Index Tables. There is a one-to-one correspondence between the entries of the index part and the entries of the second part. The

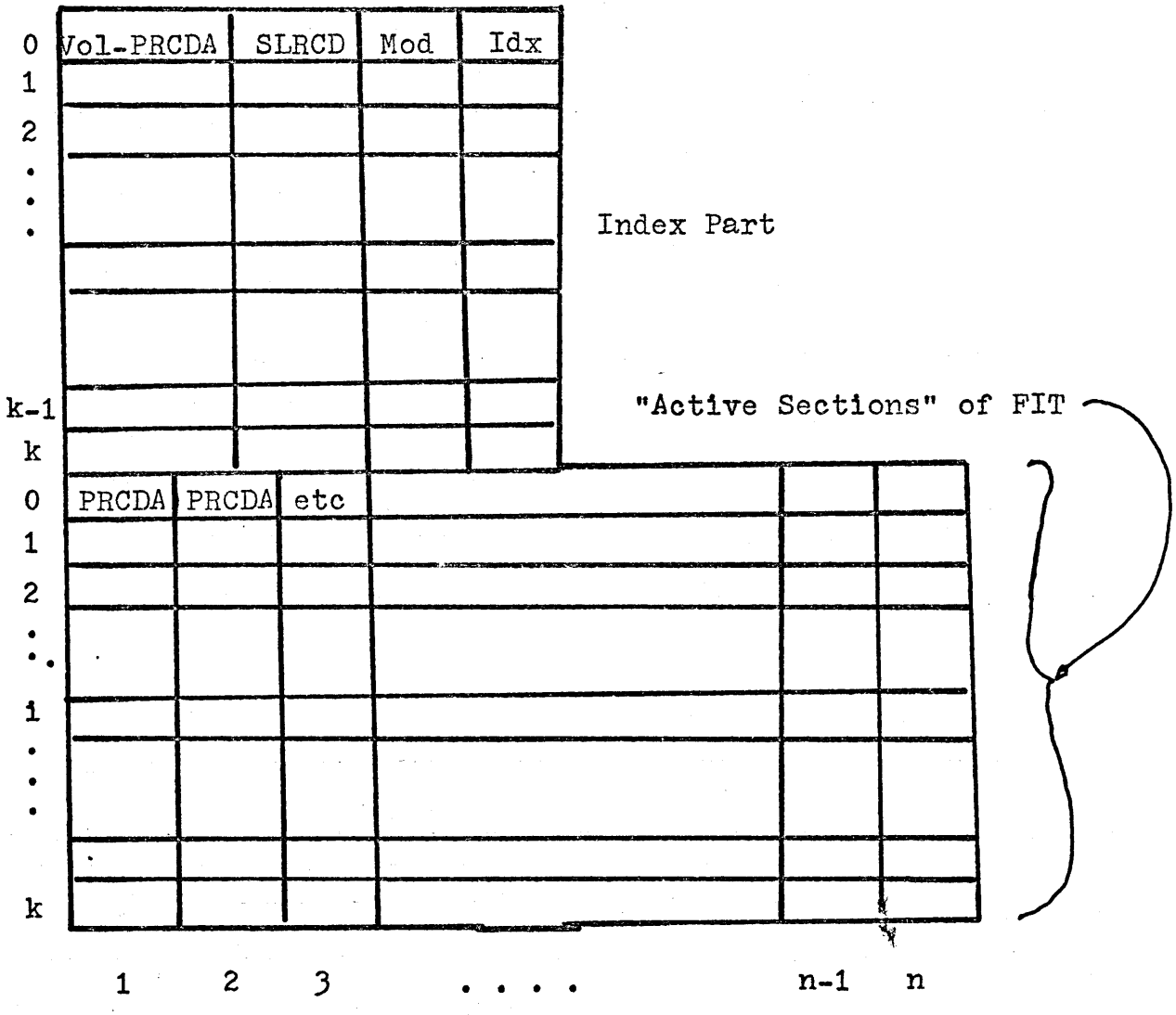


Figure 4.6--Structure of Active File Index Table

correspondence is defined by letting entry "i" of the index part specify entry "i" of the second part.

An entry of the second part contains a contiguous collection of logical record entries from the File Index Table uniquely identified by the volume and physical record address contained in the first field of the index entry. The first logical record contained in the contiguous collection of logical records of an entry is given by the second field of the index entry. The Mod field of an index entry is turned on when physical records are allocated or deallocated to the logical records contained in the entry to which it refers. The age field of an index entry contains the "age" of the indexed entry. The "age" is a function of the frequency an entry is accessed relative to the other entries. An entry which is referenced frequently would be "younger" than an entry that was seldom referenced but had been in the Active File Index Table a long time. The Index (Idx) field contains the index part of the unique file identifier. This is required in the AFIT to insure that all physical records assigned to a FIT are updated when a file is "closed."

As space is required in the Active File Index Table for sections of the mapping functions of the other File Index Tables, the entries having the "greatest age" are transferred to secondary storage if modified; else, they are simply replaced by the new entries.

The index part of the Active File Index Table provides an efficient means to access the physical record address contained in the logical record entries. After finding the the appropriate entry, i, in the index part, the address of logical record "X" relative to the beginning of the Active File Index Table can be calculated from Equation 4.1.

Relative Address = i * + (X - SLRCD) + Constant

Equation 4.1

The constant term is the total number of words required for the index part of the table. For the meaning of the other variables see Figure 4.6.

The size of the complete Active File Index Table can be limited to 320 words for used on the IBM 1130 System and still provides efficient mapping capabilities. Recall that for a 320-word physical record, each physical record of the File Index Table contains the mapping function for 319 logical records of a file's virtual memory. Divide these 319 logical records into 11 contiguous sections of 29 logical records each. Each contiguous section represents 9,180 contiguous words in a file's virtual memory. Now, in 320 words of core, one can keep ten of these contiguous sections and their index entries for ten different active files. The FOSM can dynamically choose to keep more than one entry in the Active File Index for a single file if more than one 9,180 contiguous word sections are being actively accessed

by one or more users. In fact, the FOSM will use the Age field to keep the most active sections of the File Index Tables in the Active File Index Table in order to minimize I/O operations required to map logical records into physical records.

Mapping Logical Records into Physical Records

The logical Record List is used by the FOSM to drive another routine called Prepare Physical Record List (PPRL) which maps logical records into physical records using the Active File Index Table. The format of the Physical Record List is shown in Figure 4.7. After the FOSM completely prepares the Physical Record List, the Device Strategy Module is called, with the Physical Record List as an argument, to transfer the request between core storage and secondary storage.

	n	
1	Vol-PRCDA	Core Addr
2	Vol-PRCDA	Core Addr
.		
.		
.		
n-1		
n	Vol-PRCDA	Core Addr

Figure 4.7-Physical Record List (PRCDL)

There may not be enough space in the Physical Record List to contain all the logical records in a particular request. In such cases, a user's request will be accessed in units of n-requests. The Logical Record List will always indicate the extent to which a particular request has been processed.

After the logical records have been mapped into physical records, the FOSM sequences through the Physical Record List one entry at a time to see if the requested physical record is in the I/O buffers. For physical records which are contained in the I/O buffers, the FOSM directly transfers the requested physical record or parts thereof between the I/O buffers and the stipulated area in core storage. The Logical Record List is used to determine the number of words and the starting word in each physical record of a request contained in a buffer. For each physical record directly transferred by the FOSM, the corresponding entry in the Physical Record List is deleted.

Next, the FOSM assigns buffers for the "partial" physical records contained in the request. The first and third entries of the Logical Record List are used to determine if buffers are required. If buffers are assigned, the Core Address field of the appropriate entry in the Physical Record List is changed to the address of the assigned buffer. After transferring the information into

the assigned buffer, the Logical Record List contains all the information necessary to transfer the correct part of the physical record to the user's area in core. For each read or write request to the FOSM, there are at most two I/O buffers required, one for the leading partial logical record and one for the lagging partial logical record. Either or both of these record conditions may be absent in any particular request. If two buffers are required to completely transfer a request, the FOSM attempts to assign two buffers. If only one buffer is available, the request is divided into two parts and processed sequentially.

The buffers available for the FOSM to use for transferring partial record requests between core and secondary storage are maintained in a Buffer Control Table. The format of the Buffer Control Table is shown in Figure 4.8.

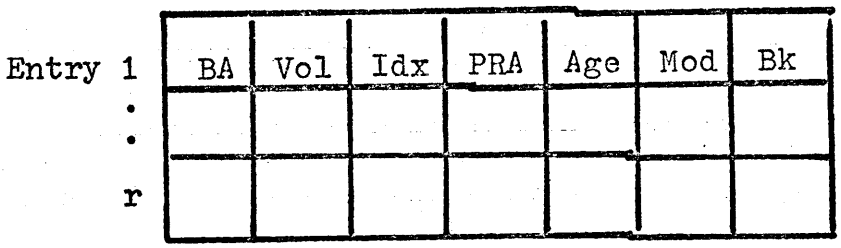


Figure 4.8--Format of Buffer Control Table

The fields of each entry specify, respectively, the available buffer address (BA), volume (Vol) and index (Idx) of the file in the buffers, physical record address (PRA)

of the record in a buffer, age (Age) of the record entry in the Buffer Control Table, whether or not the entry is modified (Mod), and if the entry is blocked (Bk). An entry is blocked from the time it is assigned to a record of a file until after the record has been transferred to the buffer. If a definite request is made for a buffer when all entries in the Buffer Control Table are blocked and the Storage Management System can not provide a new buffer, the user's process associated with the buffer request is blocked until a buffer is available.

By utilizing available buffers for partial records of the file as well as file maps, the FOSM may substantially reduce the number of I/O operations for file accesses.

Algorithms of File Organization Strategy Module

The logical flowcharts of the submodules and routines contained in the FOSM are given in Appendix C. The arguments required by these submodules and routines are presented with the flowcharts.

CHAPTER V

ALLOCATION STRATEGY MODULE

Functional Description

The function of an Allocation Strategy Module (ASM) is to find and allocate physical records for a file that is being created or expanded. It is also the responsibility of an ASM to deallocate or free the physical records assigned to a file when the file is deleted or truncated.

Design Considerations

The particular allocation strategy chosen to assign physical records to a file should be closely correlated with the file's organization and hence to a particular File Organization Strategy Module in order to achieve the intended performance of the overall system design. In fact, different File Organization Strategy Modules may require distinct allocation strategies.

The amount of core storage required for allocation information should be kept as small as possible while the number of I/O operations is minimal.

The design of an Allocation Strategy Module to support the File Organization Module, already discussed, will now be presented. The concepts and techniques used in this design can be readily modified to produce other tailored

Allocation Strategy Modules.

Design of Allocation Strategy Module

A bit map is associated with each physical volume. The Volume Bit Map (VBM) defines a function which associates each physical record on a volume with a bit position within the bit map. Bit 0 corresponds to physical record 0, bit 1 to physical record 1, etc. If a bit is set to 0, the corresponding physical reocrd is available for allocation. When a physical record is allocated to a file, the corresponding bit is set to 1. The VBM provides a relatively small space within which to represent the allocation information required by the Allocation Strategy Module. However, for a file system with several volumes, each containing hundreds of physical records, the ASM may not have enough core storage available to keep all the compact VDM's core resident.

To overcome the problem, the Allocation Bit Map can be subdivided into contiguous segments. A segment from each ABM can be maintained in a core resident table for easy access. The core resident table will be called the Active Allocation Bit Map (AABM). The format of a typical entry within the AABM is shown in Figure 5.1. The information contained within such an entry is particulary useful for volumes mounted on direct access devices.

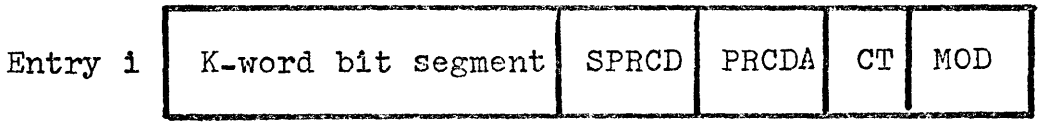


Figure 5.1-Format of Entry within Active Allocation Bit Map

Entry 0 corresponds to volume 0, entry 1 to volume 1, etc. The number of words, K, in the bit segment will depend on the number of physical volumes and the amount of core storage available for the AABM. The Starting Physical Record (SPRCD) field stipulates the physical record number that corresponds to the first bit of the k-word bit segment. This field is required to effectively calculate which physical record corresponds to which bit position in the bit segment. The Physical Record Address (PRCDA) field contains the physical record address of the ABM. This is used to "shuttle" different segements of the ABM to and from the AABM. The count (CT) field of entry "i" contains the number of physical records available for allocation on volume "i." This count may be useful in the selection of a particular volume on which to put a file. If the File Organization Strategy Module selects a volume for a file that is being created, the AABM may be a common data base for the FOSM and the ASM. An alternative is to assign the function

of selecting volumes when files are created to the ASM.

The Modification (MOD) field consists of a bit which is either on or off. The MOD bit is off until some bit position within the K-word bit segment is modified, then the bit is turned on. The MOD bit permits a means by which the modified bit segments in the AABM may be periodically transferred to the respective ABM's without having to transfer the unchanged bit segments. For many small computers which do not have protected areas in core, it is advisable to periodically update the permanent ABM's on secondary storage in order to efficiently overcome the damage which occurs when the contents of core storage are inadvertently destroyed. This is indeed a difficult problem to overcome and have the file system perform efficiently. In the design presented for the IBM 1130 computer which does not have protected areas in core, the FOSM calls the ASM to update the ABM's before it updates the File Record Mpas. This procedure never allows a physical record to be assigned to more than one file. The Close command is used to trigger the FOSM to update the the File Record Map of a particular file.

The Close command was chosen to initiate the periodic updating of data bases residing on secondary storage to minimize the relatively slow I/O operations required for updating and to localize the temporary damage, inflicted

by the destruction of the contents of core storage, to the modifications being made to Open files.

Ocassionally, the physical medium on which a physical record is stored will become defective. When a defective physical record is detected, the corresponding bit position in the ABM is turned on to give the illusion that it is allocated.

Since the ASM supports a random or direct access FOSM, the strategy incorporated into the ASM will be random in the following sense. When the FOSM calls the ASM to allocate a physical record for a file on a particular volume, the ASM will scan the appropriate bit-segment in its AABM until it encounters a bit position which is available for allocation. This bit position will be set to 1 to indicate it has been allocated to some file and the corresponding physical record number will be calculated for return to the FOSM. The FOSM calls the ASM to deallocate a set of physical records when a file is deleted or truncated. This strategy allows the ASM to minimize the number of I/O operations required to access the different bit-segments which contain the bit positions corresponding to the physical records in the deallocated request.

Algorithms of the ASM

The detailed logical flowcharts of each of the submodules making up the ASM are given in Appendix D. The arguments required by each of these submodules are found with each flowchart.

CHAPTER VI

DEVICE STRATEGY MODULE

The Device Strategy Module converts a set of I/O requests from the FOSM and the ASM into actual machine I/O command sequences.

The design of the DSM is extremely dependent on the characteristics of the I/O devices and on the I/O controller, within the Monitor, which coordinates all physical I/O on the computer system.

A Device Strategy Module which interfaces with the direct access, moveable head, disk devices of the IBM 1130 is described. This Device Strategy Module performs its function within an I/O environment provided by an I/O Controller like the one discussed in C. H. Hollander's recent MIT Thesis on a multi-tasking monitor for the IBM 1130 computer (Hollander 69).

Design of a Device Strategy Module

The FOSM calls the DSM with a list of I/O requests associated with a particular user such as read physical record 600 of volume 1 into core location 2000, read physical record 200 of volume 1 into core location 2320, and read physical record 601 into core location 2640.

The physical records are ordered, on the disk volumes

of the IBM 1130, into cylinders and tracks. The disk device has a movable read/write access head which moves across the surface of the disk volume perpendicular to the cylinders; it traces out a set of tracks on the disk volume as the volume rotates. Each track contains an ordered subset of the set of physical records.

In order to physically access physical records 600, 200, and 601 in the above example of a read request, the accessing head must be positioned over the cylinders containing physical records 600, 200, and 601 respectively before the read command is issued. In order to minimize the time consuming back and forth motion seeks of the accessing head for requests of this type, the DSM sorts the list of I/O requests from the FOSM into a new list having physical records in ascending order. When the DSM issues the machine I/O commands to access the individual physical records in the new list, it processes the list in a "top down" or "bottom up" manner depending on the actual position of the accessing head when the process commences.

Interaction with the I/O Controller

In order for the DSM to issue I/O channel commands to a disk device under the scheme incorporated into the I/O

Controller,* an ATTACH call for a particular disk device is made to the I/O Controller. The arguments of the ATTACH call are contained within a Device Control Block prepared by the DSM. The device identification, address of interrupt processing routine provided by the DSM, and user identification are included in the Device Control Block. The I/O Controller determines if the device requested by the DSM through the Device Control Block is currently "owned" by any user. If the device is not currently owned, it will be assigned to the user stipulated in the Device Control Block. If the device is owned by some user, this I/O Controller adds the Device Control Block to a queue which it maintains for each disk device. The I/O Controller will notify the DSM when the disk device is assigned to the user in the Device Control Block.

After a successful ATTACH has been made to a particular disk device, the DSM initializes the interrupt processing routines for that device, issues the first I/O channel command, and prepares to return to the FOSM or ASM. The interrupt processing routines issue the remaining I/O channel commands for all the requests in the request list. After each I/O channel command is completed,

* For a more detailed discussion of the requirements for user interaction with the I/O Controller, see C. R. Hollander's Thesis (Hollander 69).

the device issues a hardware interrupt which signals the I/O Controller to transfer control to the interrupt processing routine, for that device, provided by the DSM. The interrupt processing routine services the interrupt and returns control to the I/O Controller. When the last interrupt associated with the list of requests occurs, the asychronous switches are turned on to indicate that the I/O requests have been completed and the device is DETACHED.

Algorithms

The logical flowcharts for the Device Strategy Module and the interrupt processing routines are given in Appendix E.

APPENDIX A

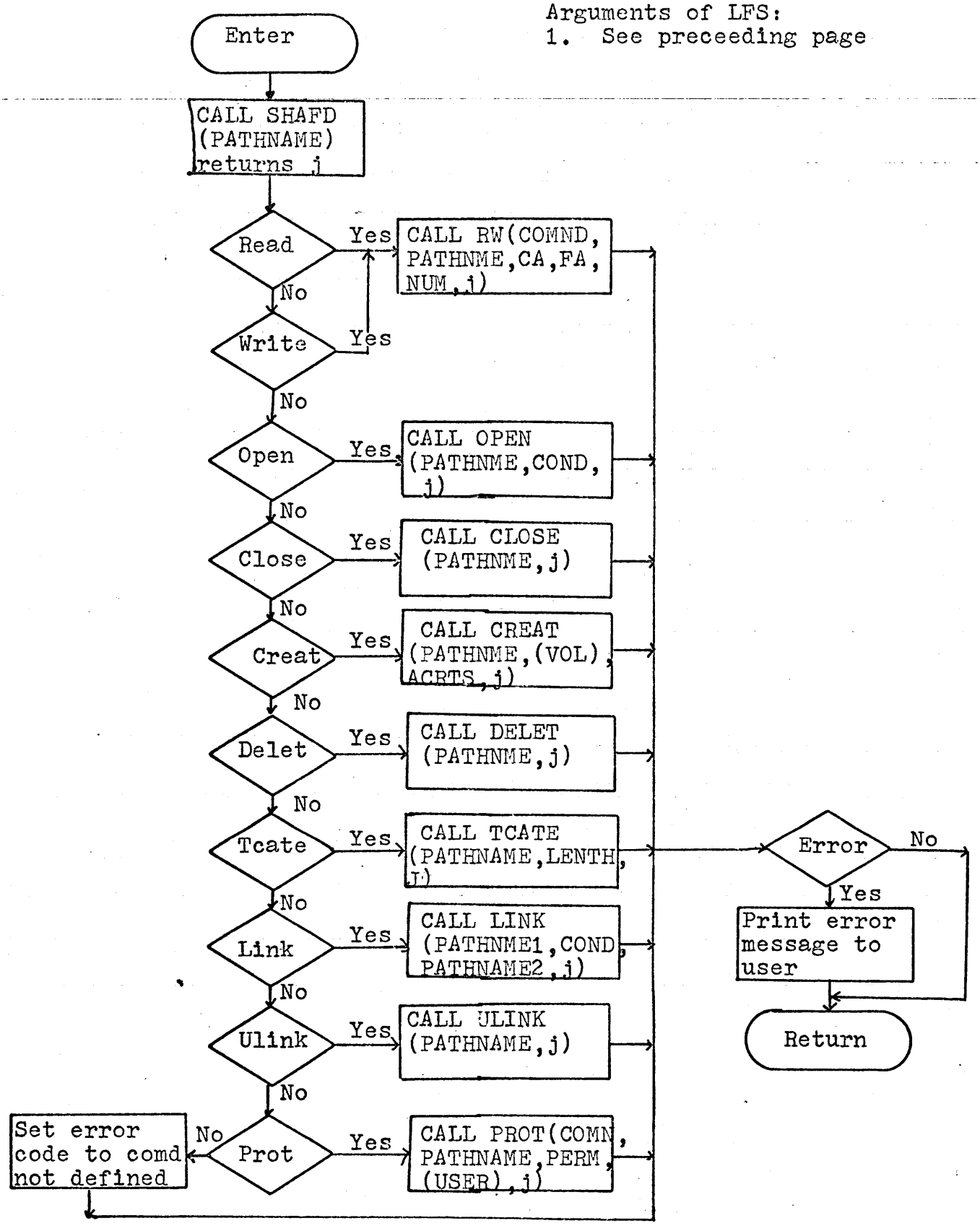
This appendix contains the detailed logical flowcharts for the algorithms and the data bases for a specific design of the Logical File System for implementation on an IBM 1130 computer.

The LFS phase is called by the users of the file system. The allowable calls from the users are listed below in flowchart notation for easy comprehension. For example CALL LFS(READ,SYMBOLIC PATHNAME,CA,FA,NUM) is the flowchart notation of a CALL to the LFS to process the READ command. The arguments in mnemonic form required by the READ command are Symbolic Pathname, CA, FA, and Num. The arguments corresponding to the mnemonic forms are given with the flowchart of each command.

1. CALL LFS(READ,SYMBOLIC PATHNAME, CA,FA,NUM)
2. CALL LFS(WRITE,SYMBOLIC PATHNAME,CA,FA,NUM)
3. CALL LFS(OPEN,SYMBOLIC PATHNAME, COND.)
4. CALL LFS(CLOSE,SYMBOLIC PATHNAME)
5. CALL LFS(CREAT,SYMBOLIC PATHNAME,(VOL),ACRTS)
6. CALL LFS(DELET,SYMBOLIC PATHNAME)
7. CALL LFS(TCATE,SYMBOLIC PATHNAME,LENTH)
8. CALL LFS(LINK,SYMBOLIC PATHNAME1,SYMBOLIC PATHNAME2, COND.)
9. CALL LFS(PROT,COMND,SYMBOLIC PATHNAME,PERM,(USER))
10. CALL LFS(ULINK,SYMBOLIC PATHNAME)

LFS PHASE
FLOWCHART FOR ALGORITHM OF MAINLINE MODULE (LFS)

Arguments of LFS:
1. See preceeding page

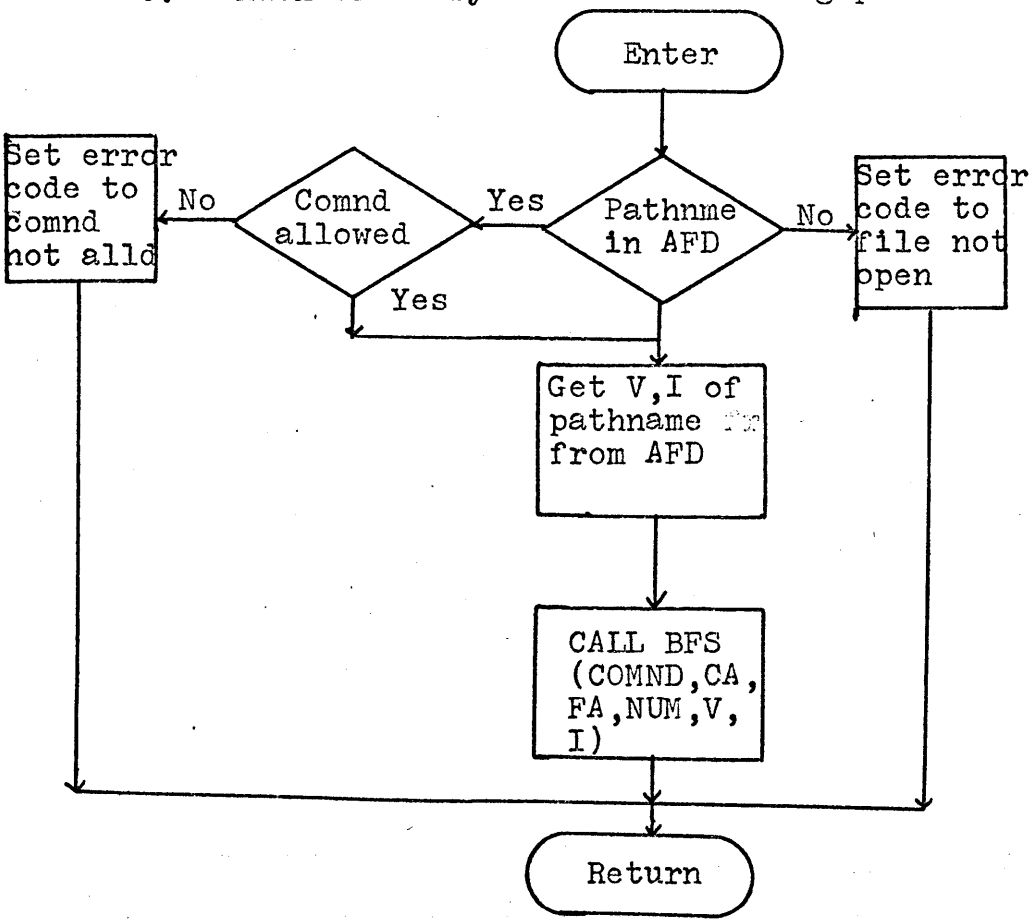


LFS PHASE

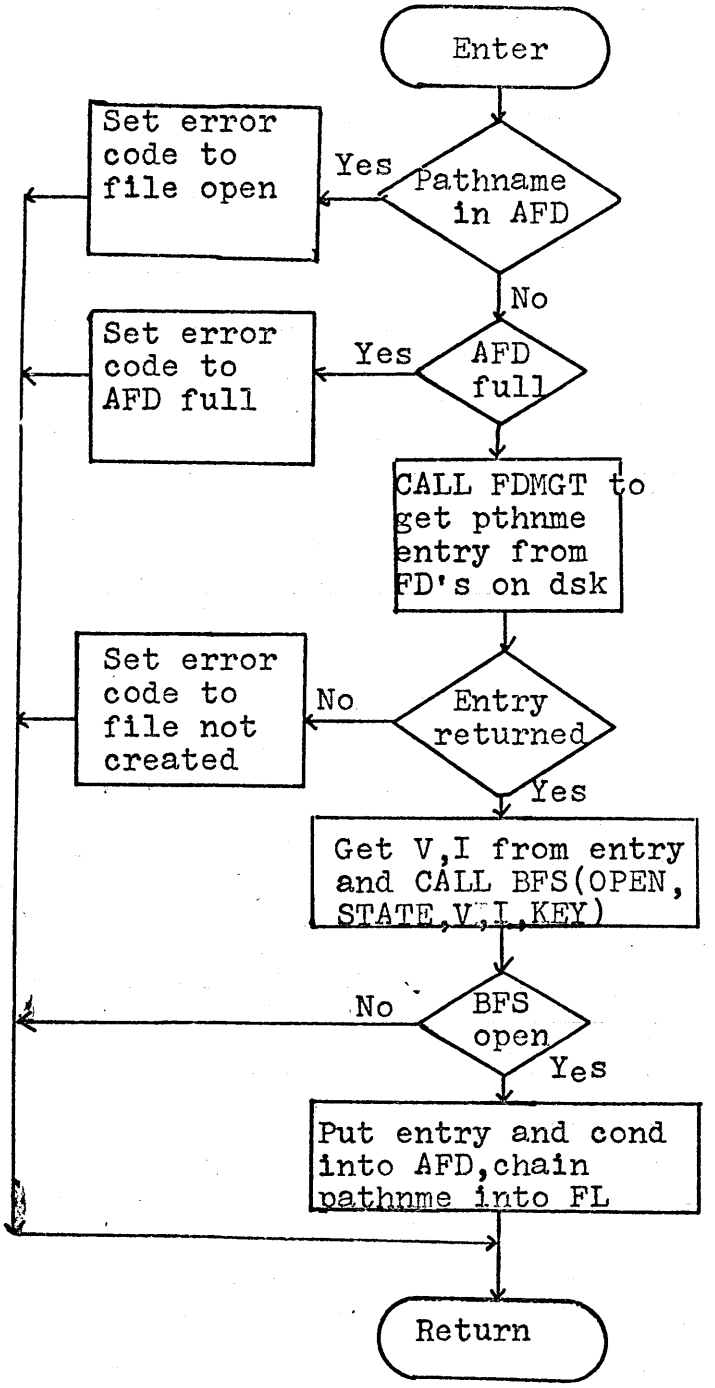
FLOWCHART FOR ALGORITHM OF READ, WRITE SUBMODULE (RW)

Arguments of RW:

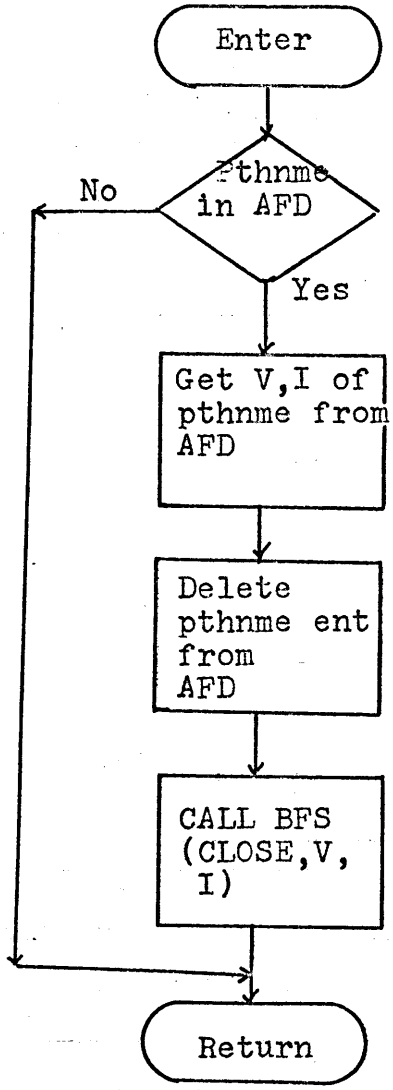
1. The Command Read or Write
2. Symbolic Pathname of a file
3. Core Address
4. File Address
5. Number of words to transfer
6. Index of entry in AFD containing pathname or -1



LFS PHASE
FLOWCHART FOR ALGORITHM OF OPEN SUBMODULE
Arguments of OPEN:
1. Symbolic Pathname of file
2. Condition for which file is to be opened
   Allowable conditions: Read, Write, Read/Write
3. Index of entry in AFD containing pathname or -1



LFS PHASE
FLOWCHART FOR AIGORITHM OF CLOSE SUBMODULE
Argument of CLOSE:
1. Symbolic Pathname of a file
2. Index of entry in AFD containing pathname or -1

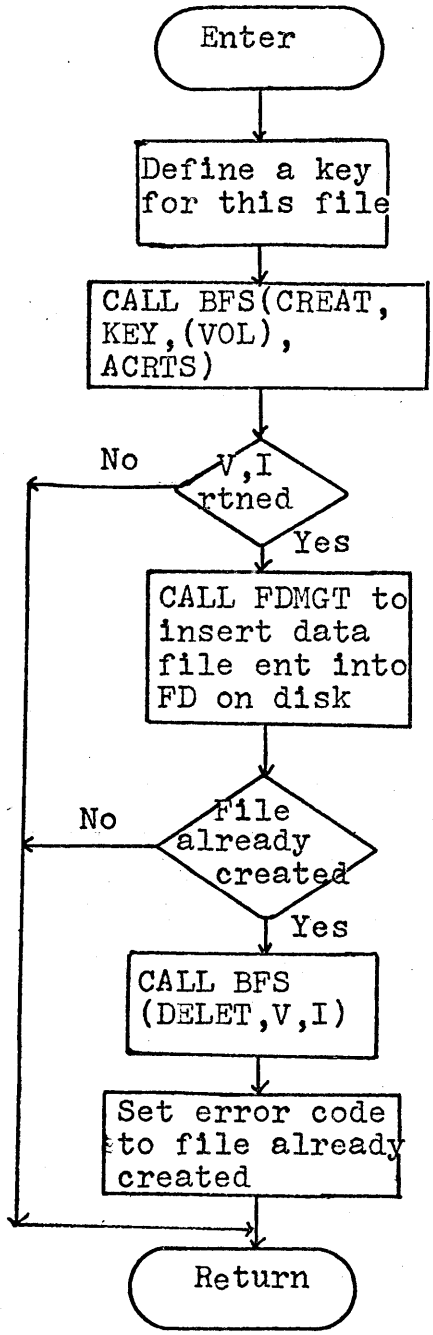


LFS PHASE

FLOWCHART FOR ALGORITHM OF CREATE SUBMODULE (CREAT)

Arguments of CREAT:

1. Symbolic Pathname of file
2. Symbolic Volume name may be given Optional
3. Access Rights specified by user to describe file
4. Index of entry in AFD containing pathname or -1

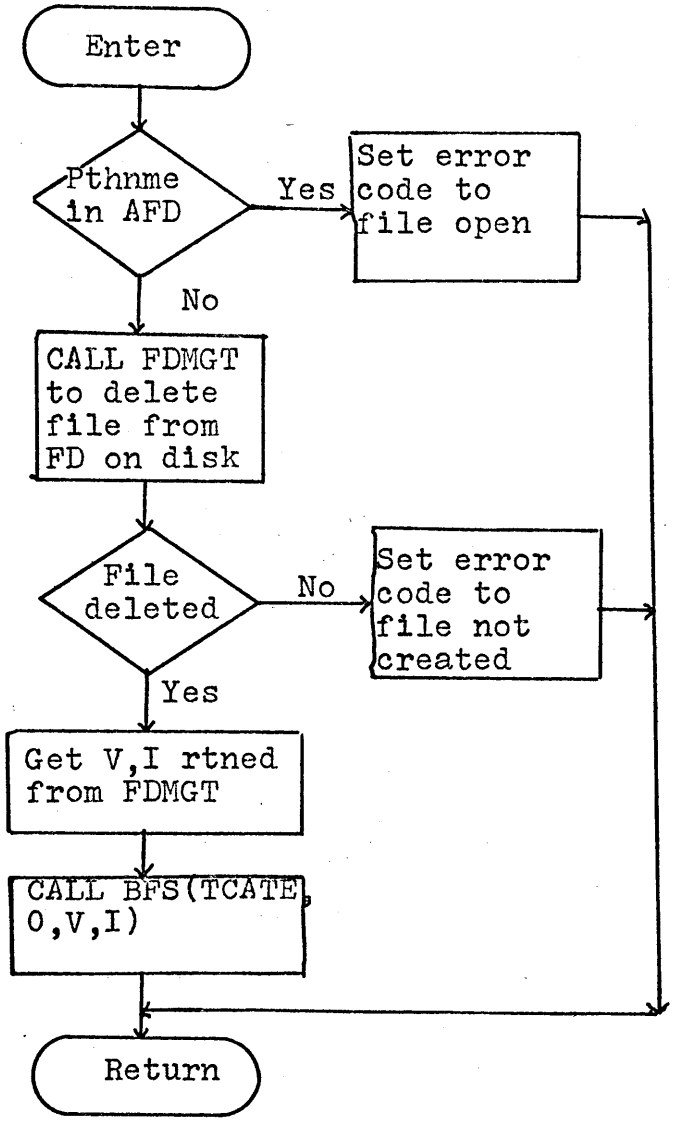


LFS PHASE

FLOWCHART FOR ALGORITHM OF DELETE SUBMODULE (DELET)

Arguments of DELET:

1. Symbolic Pathname of a file
2. Index of entry in AFD containing pathname or -1

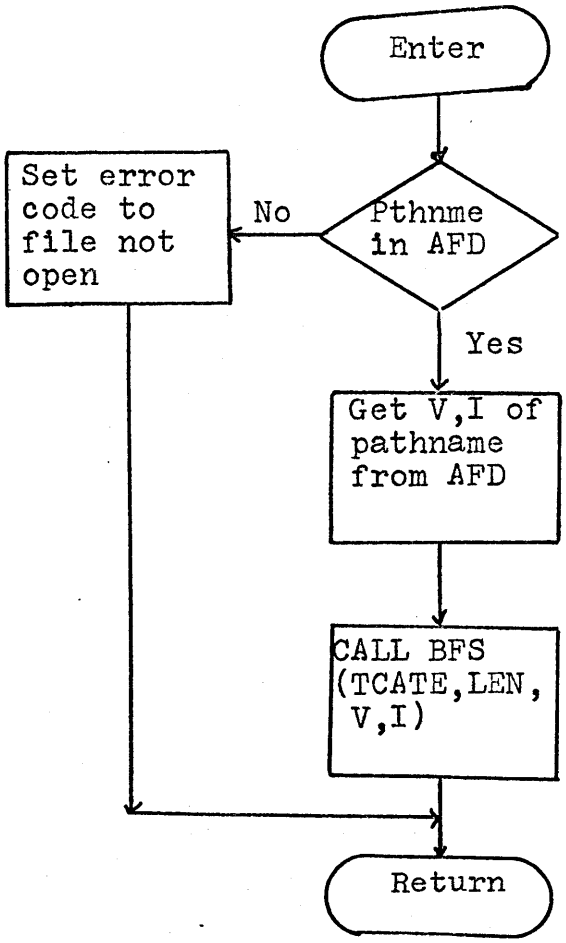


LFS PHASE

FLOWCHART FOR ALGORITHM OF TRUNCATE SUBMODULE (TCATE)

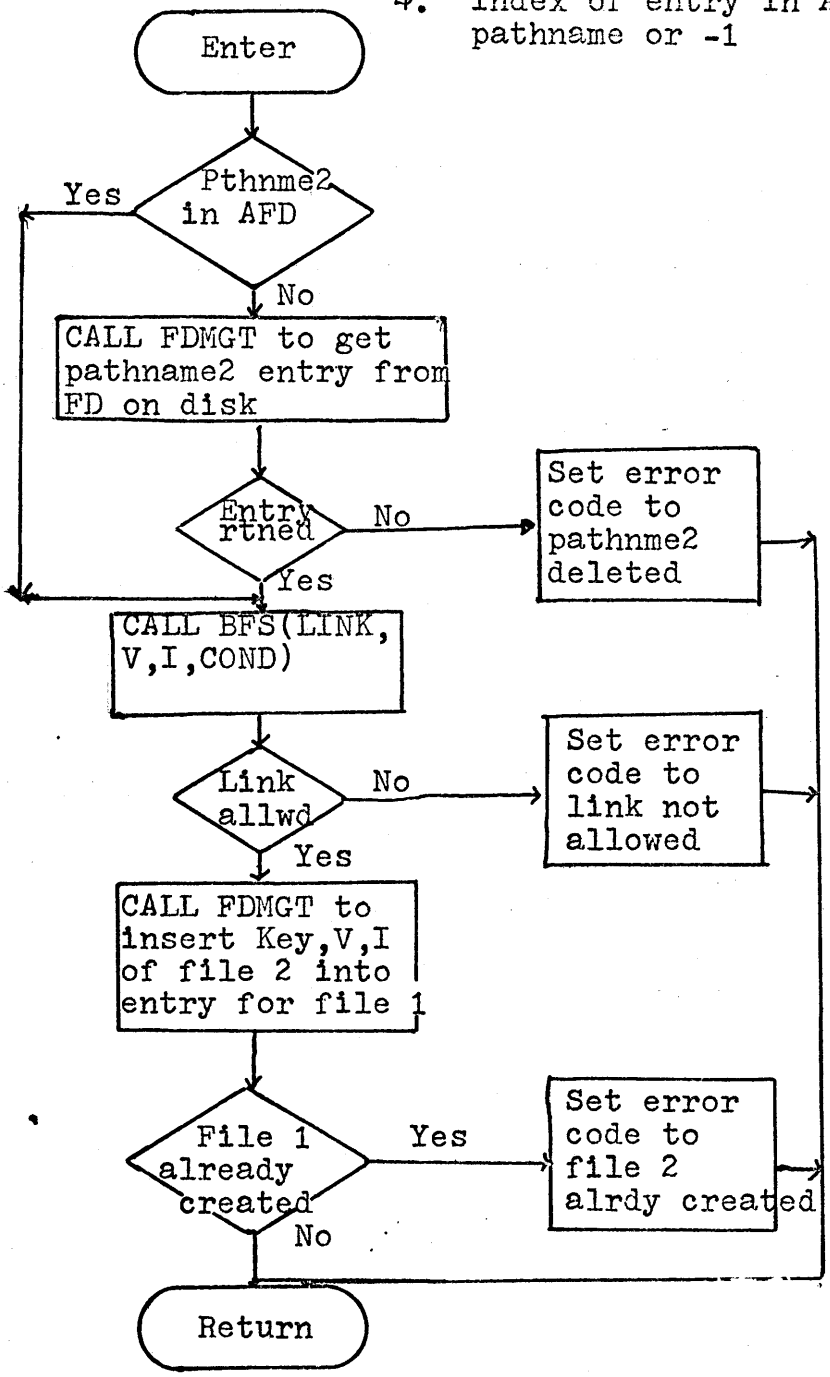
Arguments of TCATE:

1. Symbolic Pathname of a file
2. Length to which file wil be reduced
3. Index of entry in AFD containing pathname or -1



LFS PHASE
FLOWCHART FOR ALGORITHM OF LINK SUBMODULE
Arguments of LINK:

1. Symbolic Pathname of data file to be created
2. Symbolic Pathname of data file to which created file will be linked
3. Condition of link-Alowable are: Read, Write, Read/Write
4. Index of entry in AFD containing pathname or -1

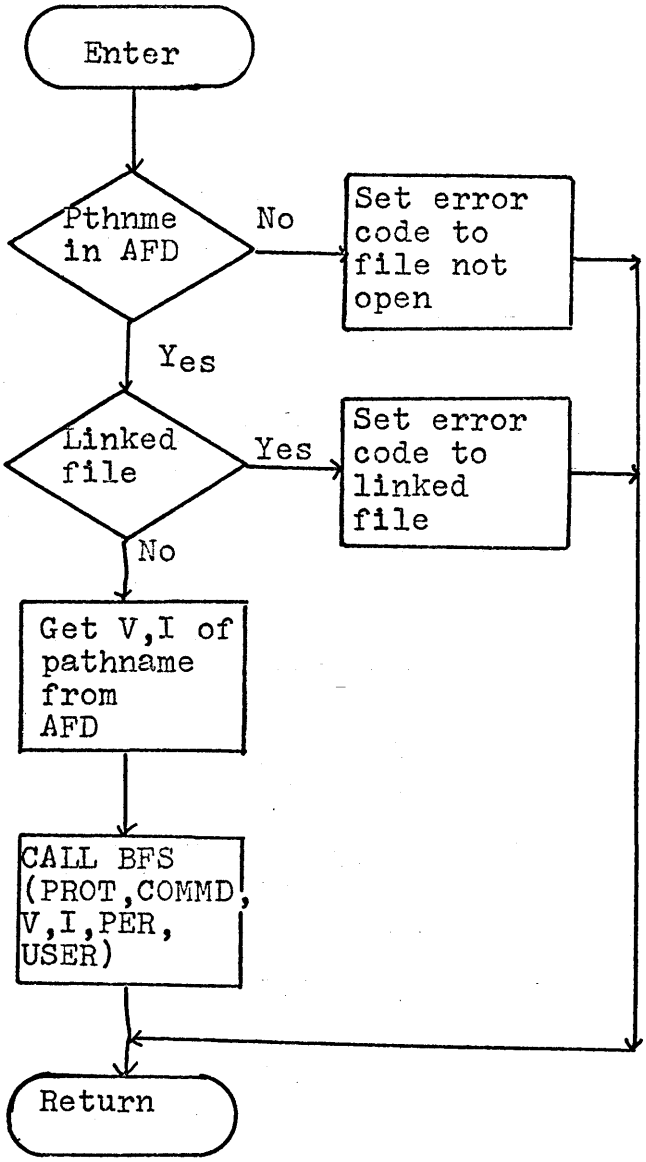


LFS PHASE

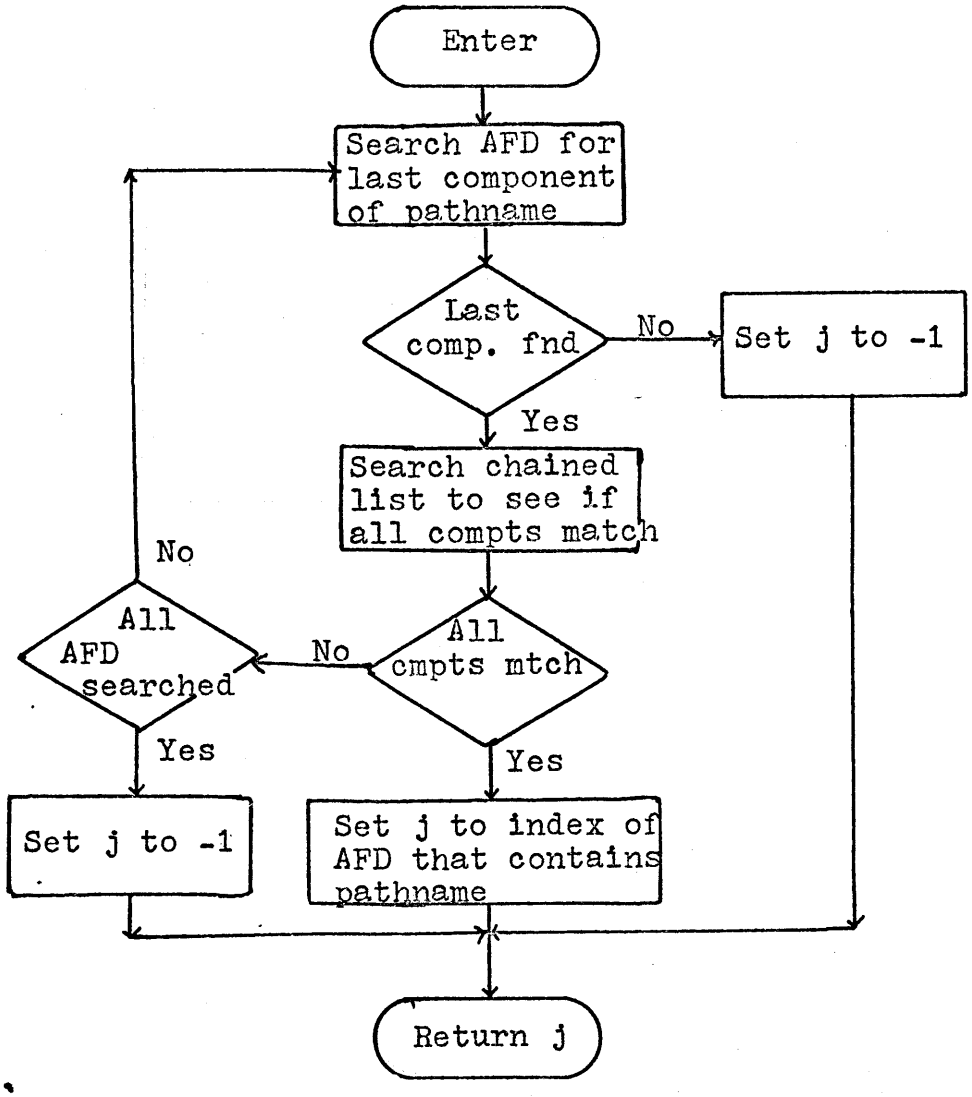
FLOWCHART FOR ALGORITHM OF PROTECT SUBMODULE (PROT)

Arguments of PROT:

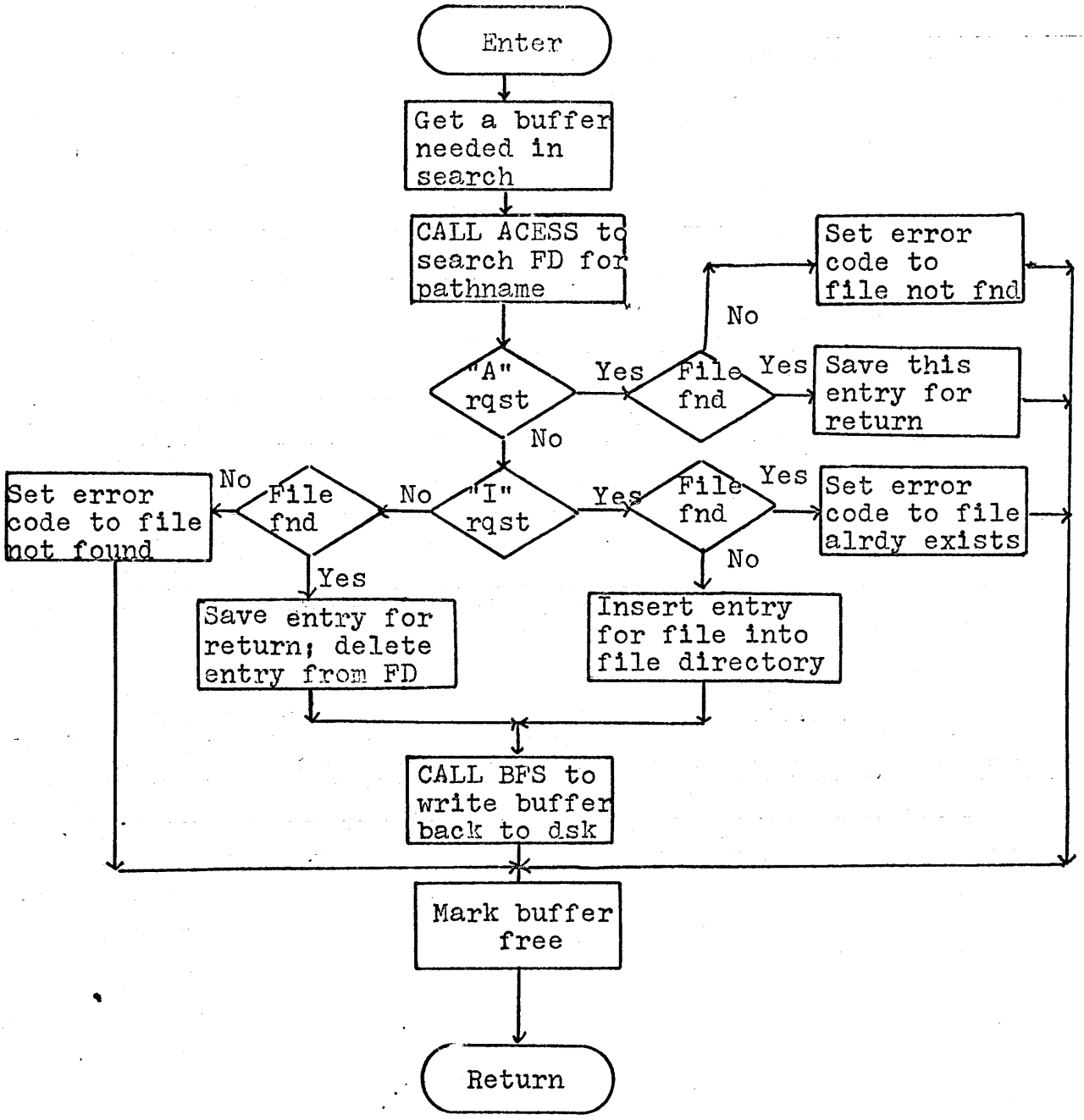
1. The Command Add or Delete
2. Symbolic Pathname of a file
3. The Permission Read, Write, Read/Write, Link
4. User associated with permission Optional
5. Index of entry in AFD containing pathname or -1



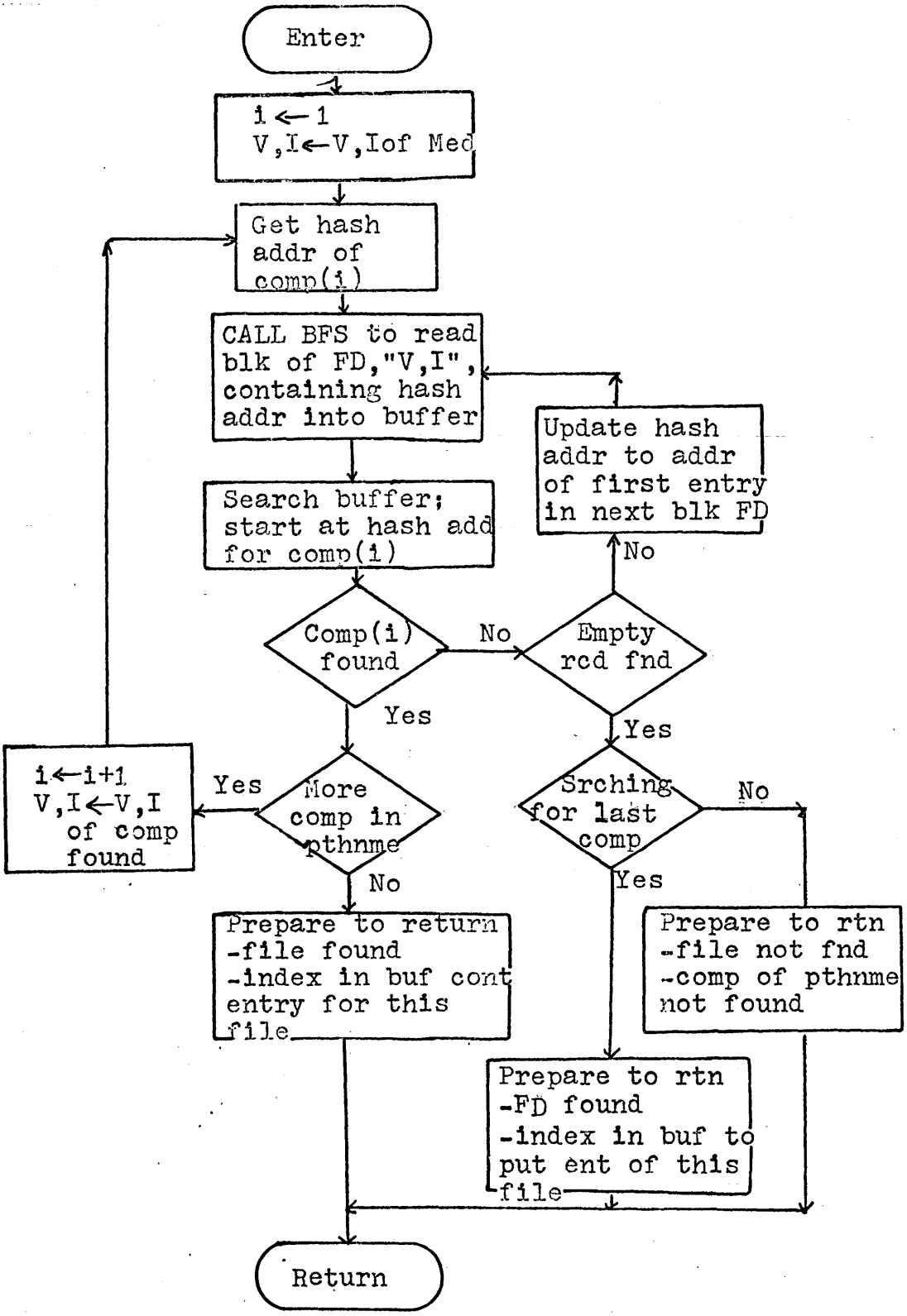
LFS PHASE
FLOWCHART OF ALGORITHM OF SHAFD
Arguments of SHAFD:
1. Symbolic Pathname of a file



LFS PHASE
FLOWCHART FOR ALGORITHM OF FDMGT
Arguments of FDMGT:
1. The Command Access,Delete, Insert (A,D,I)
2. Symbolic Pathname of file
3. Key associated with file Optional
4. Access Rights associated with file Optional



LFS PHASE
FLOWCHART FOR ALGORITHM OF ACCESS
Arguments of ACCESS:
1. Symbolic Pathname of a file
2. Address of buffer to be used in searching

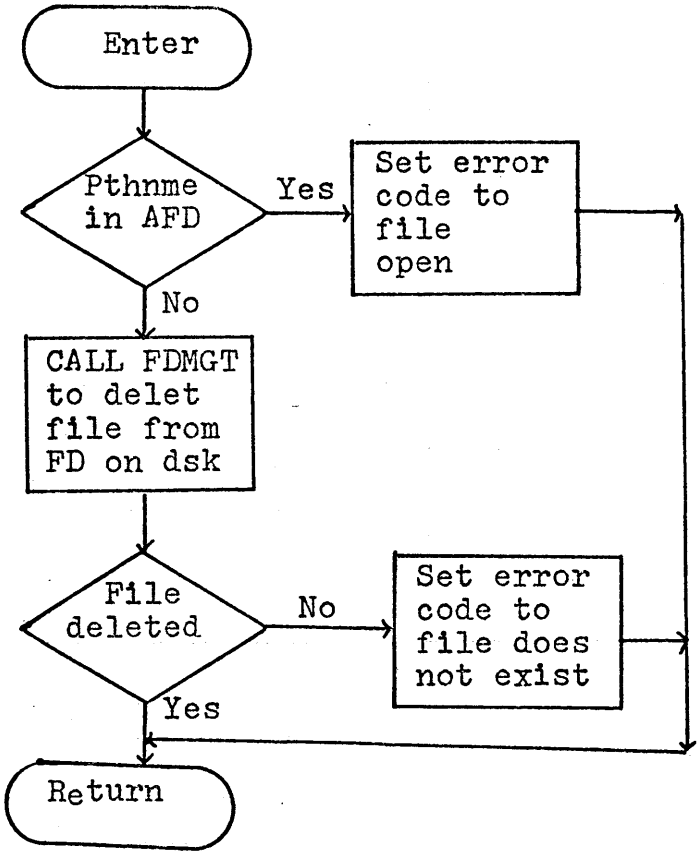


LFS-PHASE

FLOWCHART FOR AGORITHM OF UNLINK SUBMODULE (ULINK)

Arguments of ULINK:

1. Symbolic Pathname of file
2. Index of entry in AFD containing pathname or -1



DATA BASES OF LFS FOR IMPLEMENTATION ON IBM 1130 COMPUTER

Active File Directory

Entry

Symbolic	Pathname	Vol	Index	Key	ACRTS

Each entry in the AFD is eight words long. The symbolic pathname occupies four words and symbolic volume, index, key, and access rights each occupy one word of the entry. The entry for the MFD is always contained in the first row of the AFD

ACRTS

Bits	0	1	23		11	15
Subfields	A	B	C			D

Subfield

A
- 0--Unlinked file
- 1--Linked file

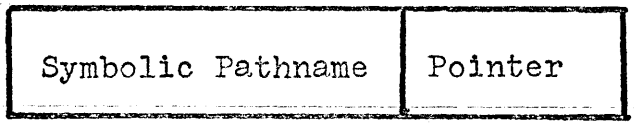
B
- 0--Data file
- 1--Directory file

C
- 01--Open for reading
- 10--Open for writing
- 11--Open for reading and writing

D
- positive integer--Index to Free List
- zero--Symbolic name contained in AFD

Free List

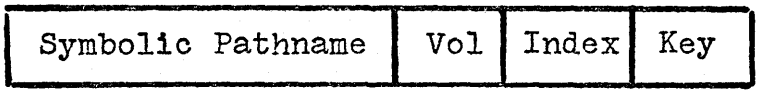
Entry



Each entry in the Free List is five words long. The first four words can contain one component of the symbolic pathname. The last word in the entry is used to chain together the entries representing a symbolic pathname.

File Directories

Entry



Each entry in a file directory is seven words long. The symbolic pathname occupies four words and the volume, index, and key each occupy one word of the entry.

APPENDIX B

This appendix contains the detailed logical flowcharts for the algorithms and data bases for a specific design of the Basic File System phase for implementaion on an IBM 1130 computer.

The BFS phase is called by the LFS phase. The allowable calls from the LFS phase are listed below in flowchart notation for comprehension. For example, CALL BFS(READ,V,I,CA,FA,NUM) is the flowchart notation of a Call command to the BFS to process the Read command. The arguments in mnemonic form required by the Read command are V, I, CA, FA, Num. The arguments corresponding to the mnemonic forms are given with the flowchart of each command.

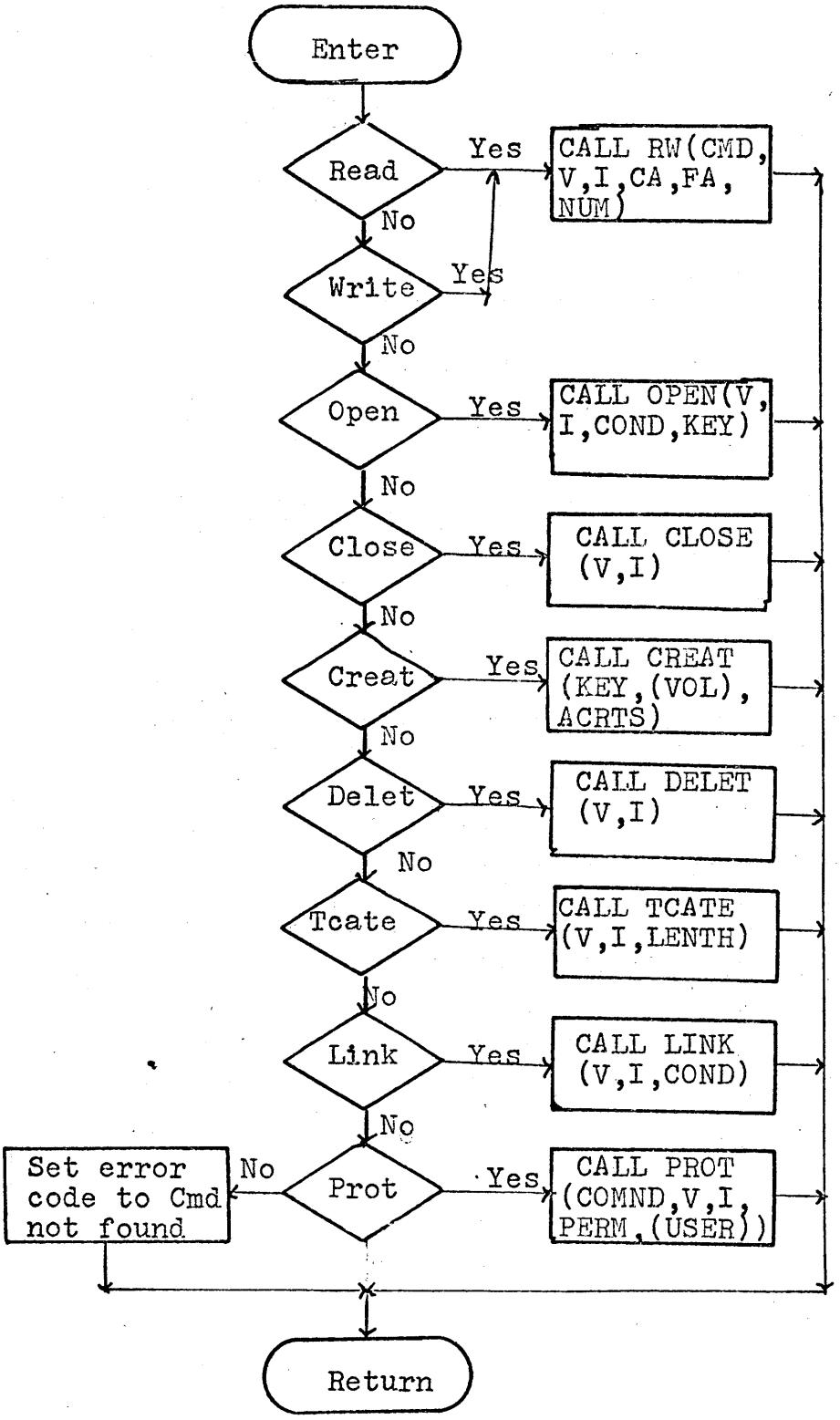
1. CALL BFS(READ,V,I,CA,FA,NUM)
2. CALL BFS(WRITE,V,I,CA,FA,NUM)
3. CALL BFS(OPEN,V,I,COND,KEY)
4. CALL BFS(CLOSE,V,I)
5. CALL BFS(CREAT,KEY,(VOL),ACRTS)
6. CALL BFS(DELET,V,I)
7. CALL BFS(TCATE,V,I,LENTH)
8. CALL BFS(LINK,V,I,COND)
9. CALL BFS(PROT,COMND,V,I,PERM,(USER))

BFS PHASE

FLOWCHART FOR ALGORITHM OF MAINLINE MODULE (BFS)

Arguments of BFS:

1. See preceeding page



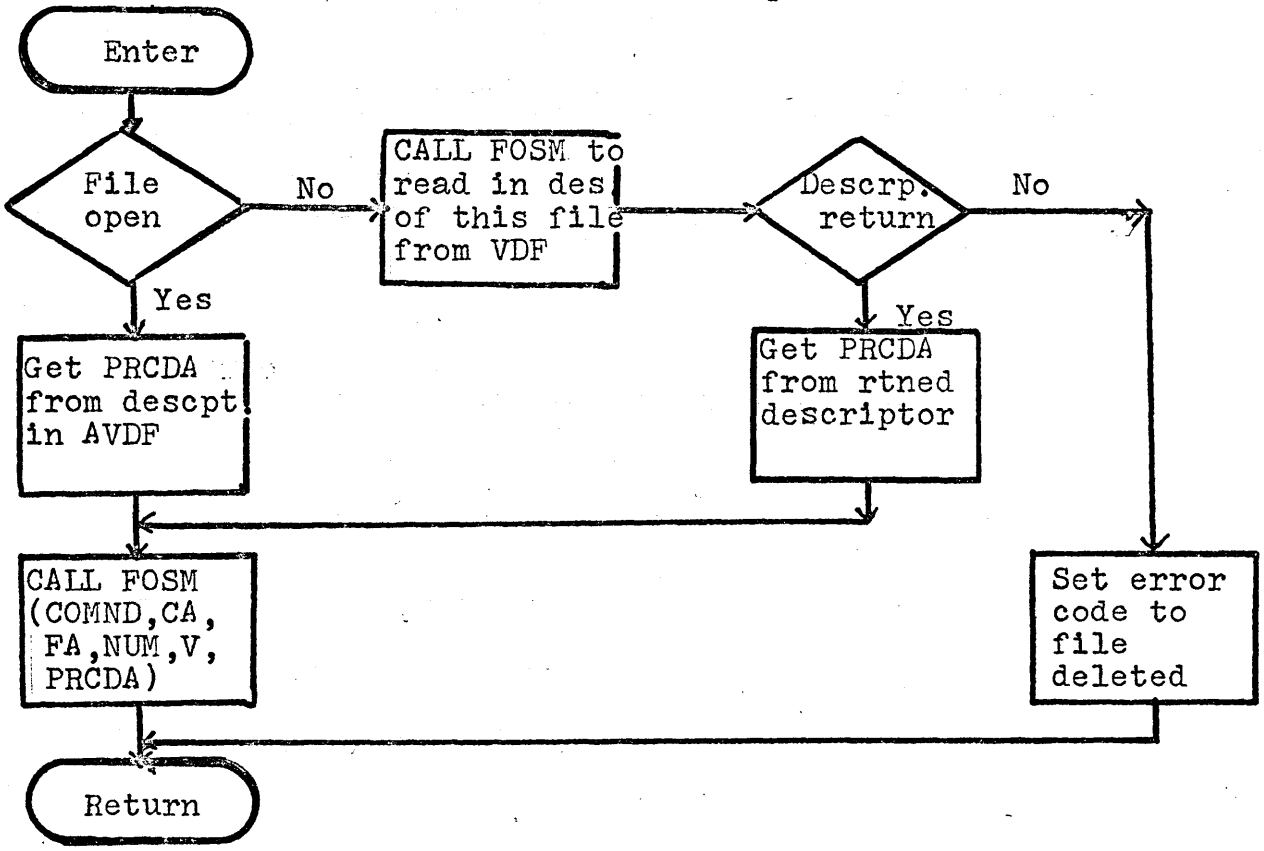
BFS PHASE

FLOWCHART FOR ALGORITHM OF READ, WRITE SUBMODULE (RW)

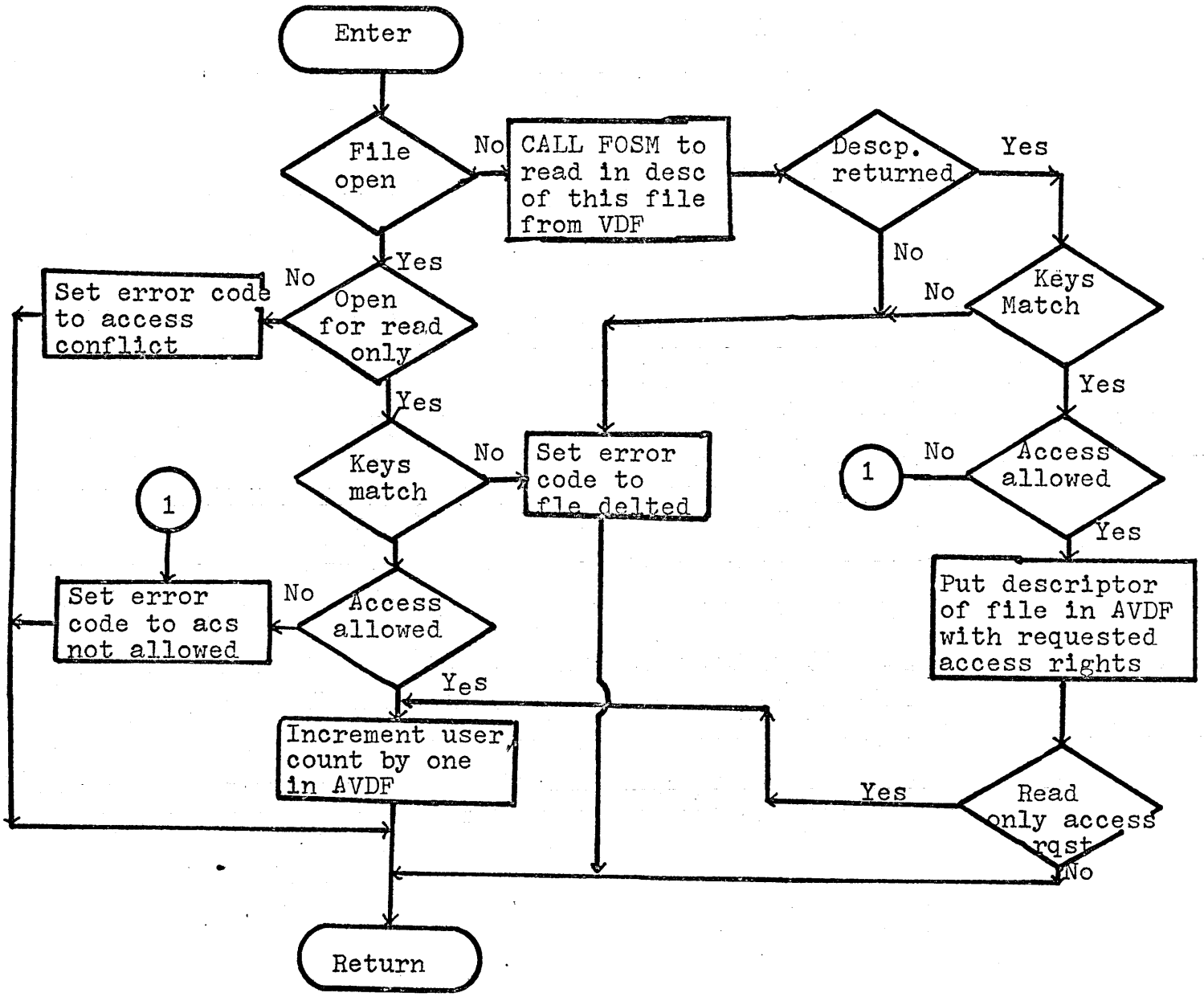
Arguments of RW:

1. The Command Read or Write
2. Unique file identifier (Vol,Index)
3. Core Address (CA)
4. File Address (FA)
5. Number of words to transfer (NUM)

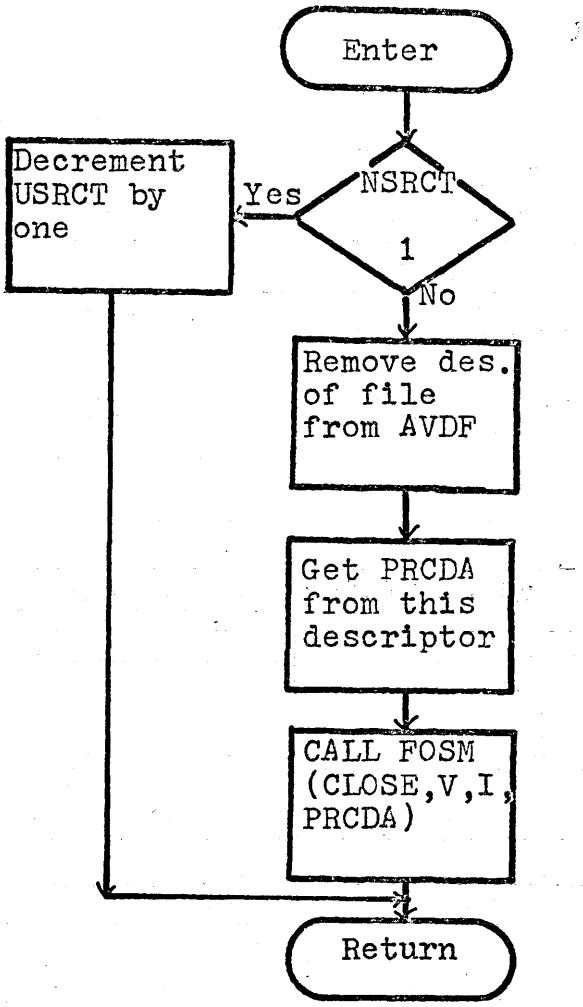
Note: This path is only used when LFS reads in directories leading to qualified name. This keeps user from having to open directory files A and B when he opens data file Y with qualified name A.B.Y.



BFS PHASE
FLOWCHART FOR ALGORITHM OF OPEN SUBMODULE
Arguments of OPEN:
1. Unique file identifier (Vol,Index) of file
2. Condition for which file is to be opened
   Allowable conditions-Read,Write,Read/Write
3. Key of file to be opened



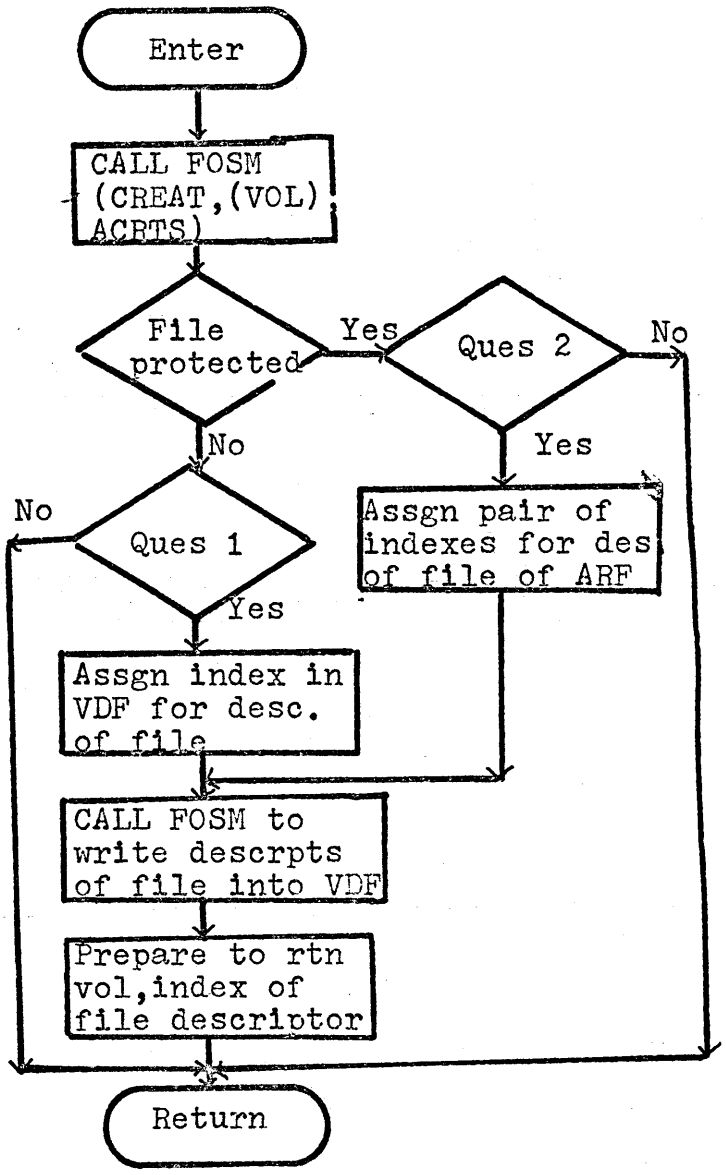
BFS PHASE
FLOWCHART FOR ALGORITHM OF CLOSE SUBMODULE
Arguments of CLOSE:
1. Unique file identifier (Vol,Index) of file



BFS PHASE

FLOWCHART FOR ALGORITHM OF CREATE SUBMODULE (CREAT)

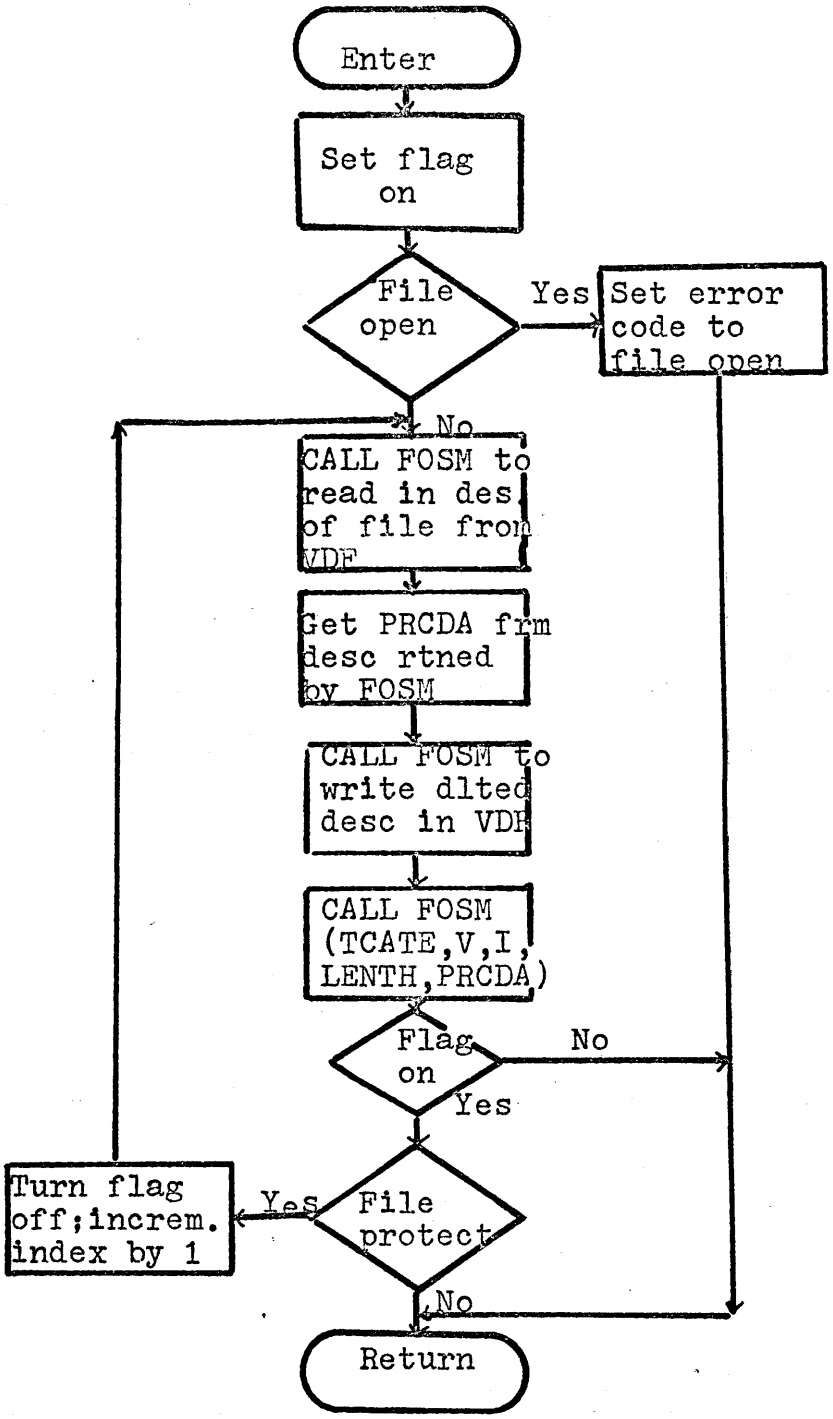
1. Key for file to be created
2. Symbolic Volume may be given Optional
3. Access Rights (ACRTS) specified by owner



Question 1?--Did FOSM return a value for volume and PRCDA for file?

Question 2?--Did FOSM return a value for volume and two PRCDA's? First PRCDA is for created file. Second PRCDA is for Access Rights File (ARF).

BFS PHASE
FLOWCHART FOR ALGORITHM OF DELETE SUBMODULE (DELET)
Arguments of DELET:
1. Unique file identifier (Vol,Index) of file

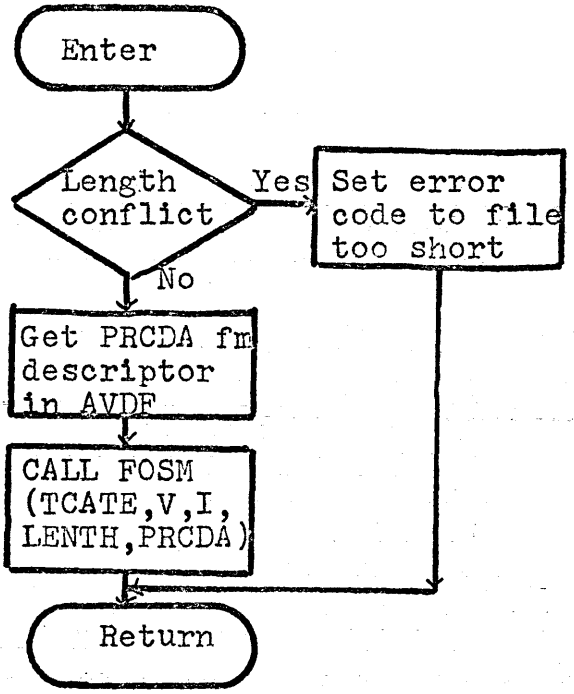


BFS PHASE

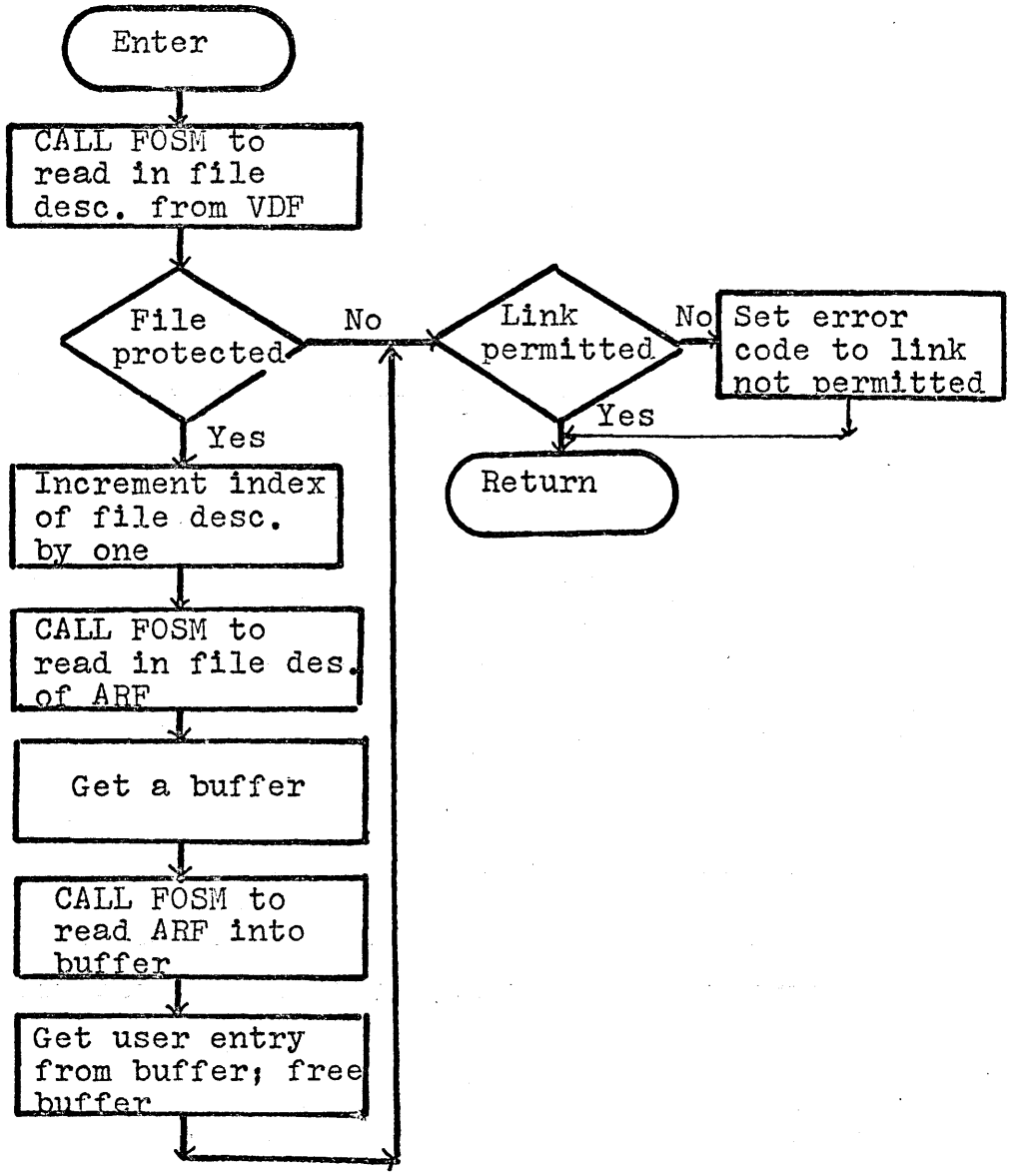
FLOWCHART FOR ALGORITHM OF TRUNCATE SUBMODULE (TCATE)

Arguments of TCATE:

1. Unique file identifier (Vol,Index) of file
2. new length (Lenth) of file



BFS PHASE
FLOWCHART FOR ALGORITHM OF LINK SUBMODULE
Arguments of LINK:
1. Unique file identifier (Vol,Index) of file
2. Condition of link
   Allowable conditions: Read,Write,Read/Write

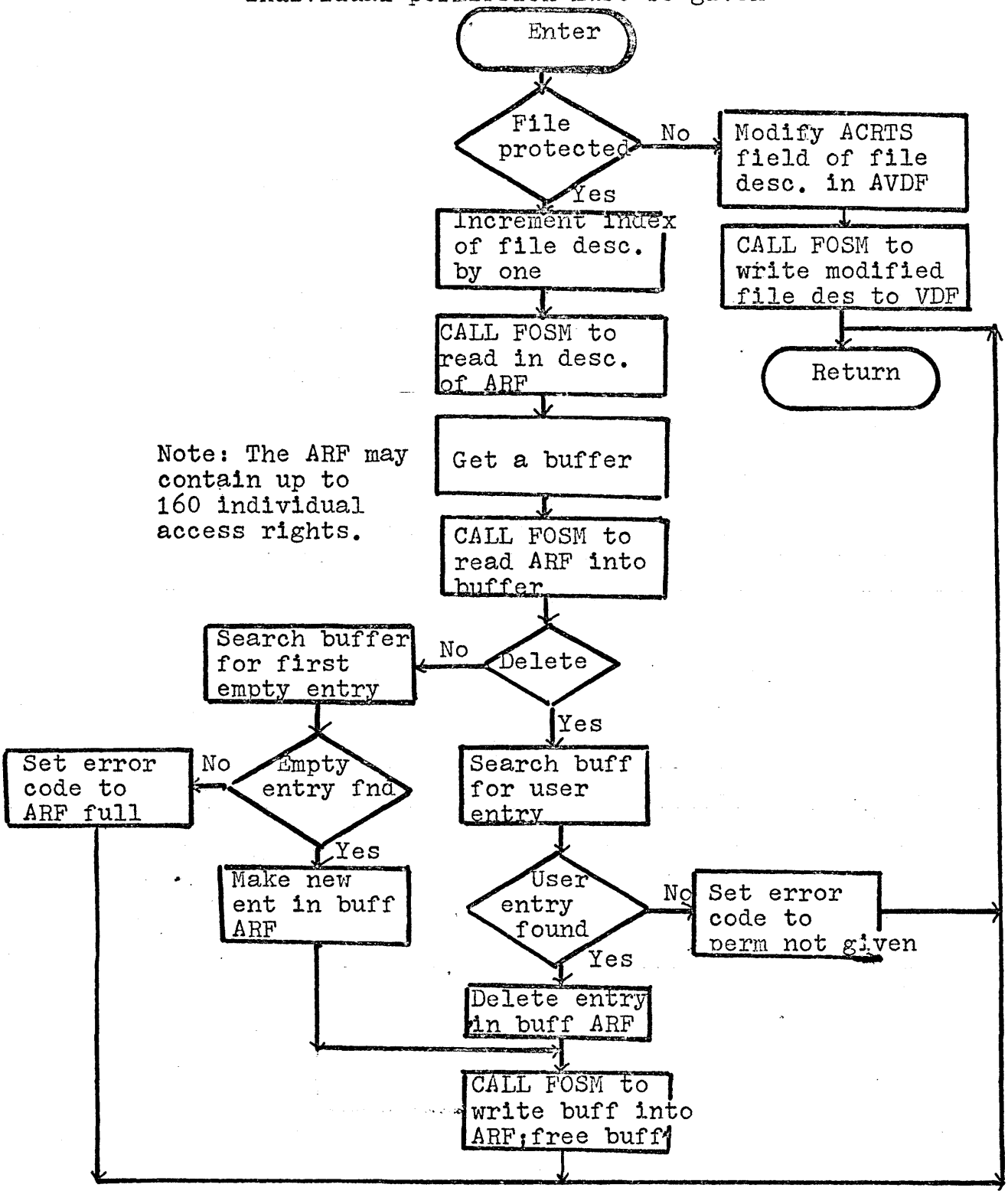


BFS PHASE

FLOWCHART FOR ALGORITHM OF PROTECT SUBMODULE (PROT)

Arguments of PROT:

1. The Command Add or Delete
2. Unique file identifier (Vol,Index) of file
3. The Permission: Read,Write,Read/Write,Link
4. If permission not global, the User associated with individual permission must be given



Note: The ARF may contain up to 160 individual access rights.

DATA BASES OF BFS FOR IMPLEMENTATION ON IBM 1130 COMPUTER

Active Volume Descriptor File

Entry	Vol	Index	PRCDA	Lenth	Key	ACRTS	USRCT

Each entry in the AVDF consists of seven one word fields. The File Organization (FO) field has been incorporated into the ACRTS field. The first R entries in the AVDF are reserved for descriptors of the Volume Descriptor Files. Since there is one VDF for each mounted volume, the exact value of R is determined by the available hardware configuration. Most IBM 1130 hardware configurations allow one, three, or five volumes to be mounted simultaneously.

ACRTS Bits	0	1,2	3,4		13-15
Subfields	A	B	C		D

Subfield

A 0--Links not allowed

1--Links allowed

B 00-Links are protected by individual user

01-Link to Read permitted by any user

10-Link to Write permitted by any user

11-Link to Read/Write permitted by any user

C 01-Read permitted by owner

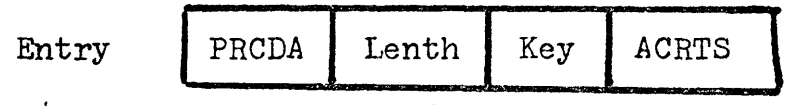
10-Write permitted by owner

11-Read/Write permitted by owner

D 0--Direct Access File Organization

1-7--Reserved for omplementation of additional Fosm's

Volume Descriptor File



Each entry in the VDF is four words long. All the fields are each one word long. The ACRTS field is subdivided as indicated for the AVDF.

APPENDIX C

This appendix contains the logical flowcharts for the algorithms and data bases for a design of a File Organization Strategy Module for implementation on an IBM 1130 computer.

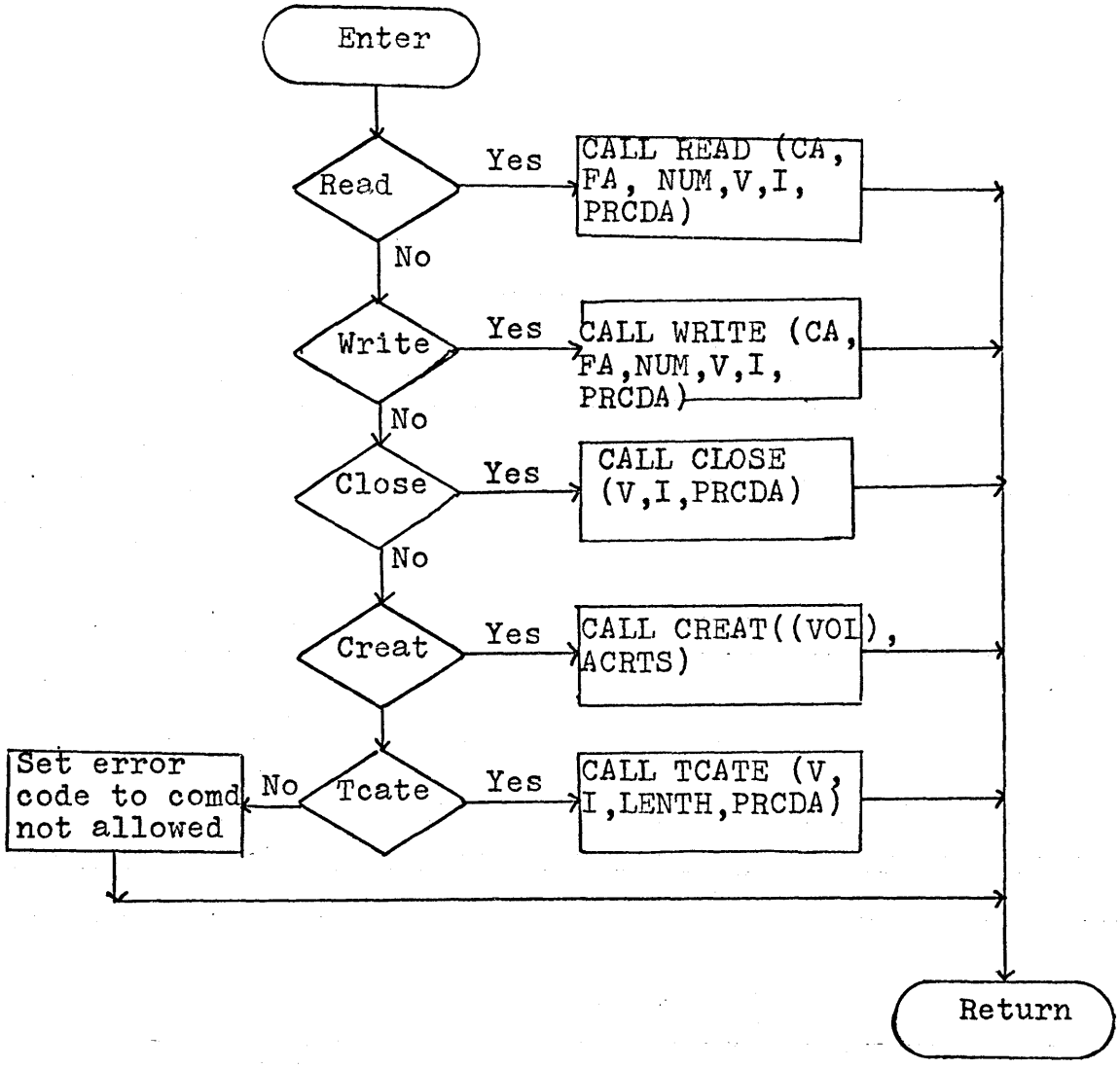
The FOSM is called by the BFS phase. The allowable calls from the BFS phase are listed below in flowchart notation as discussed in Appendix B.

1. CALL FOSM(READ,CA,FA,NUM,V,I,PRCDA)
2. CALL FOSM(WRITE,CA,FA,NUM,V,I,PRCDA)
3. CALL FOSM (CLOSE,V,I,PRCDA)
4. CALL FOSM (CREAT,(VOL),ACRTS)
5. CALL FOSM (TCATE,V,I,LENTH,PRCDA)

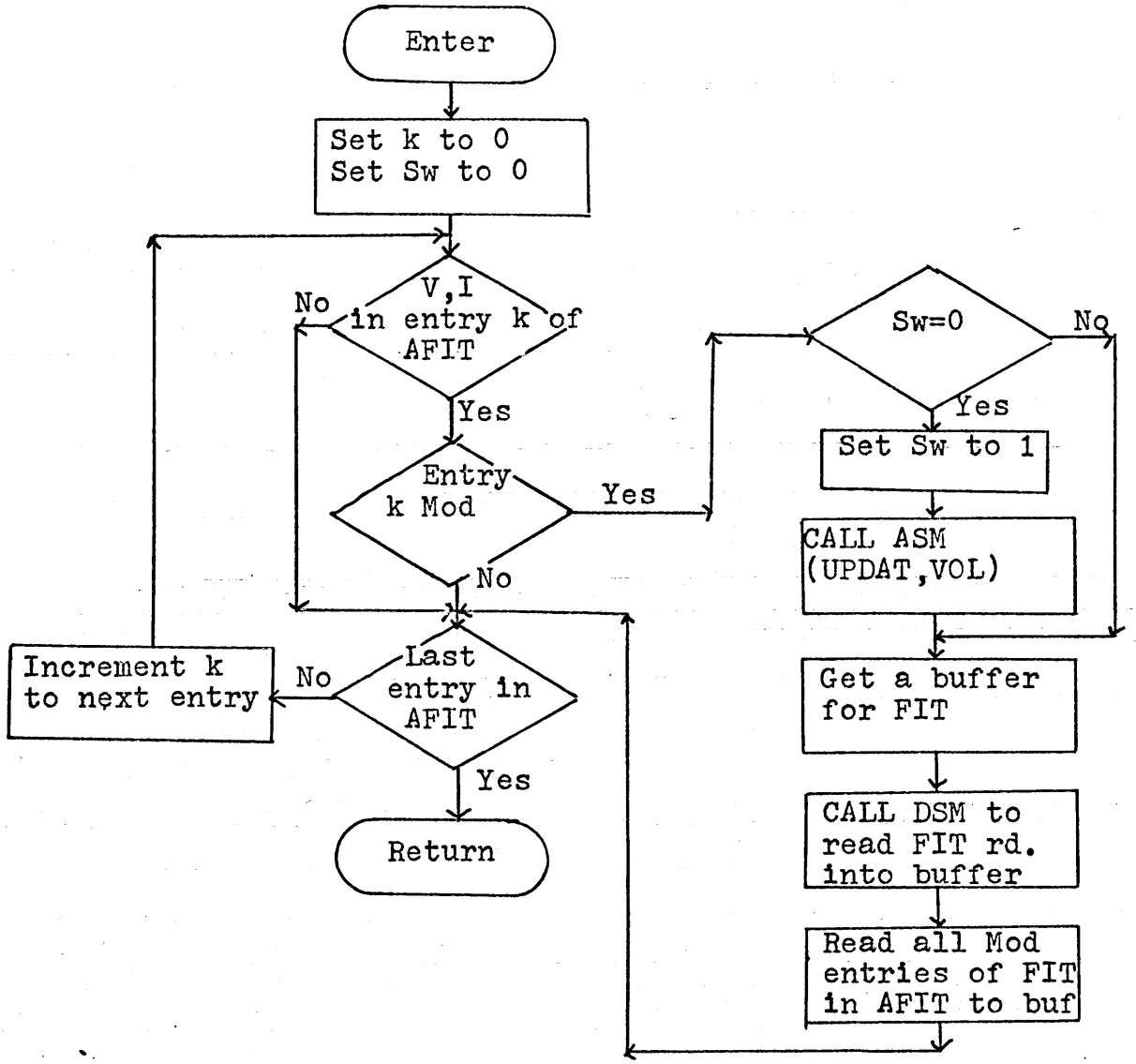
FOSM
FLOWCHARTS FOR ALGORITHMS OF FILE ORGANIZATION
STRATEGY MODULE (FOSM)

Arguments of FOSM:

1. See previous page



FOSM
FLOWCHARTS FOR ALGORITHMS OF UDFIT ROUTINE
Arguments of UDFIT:
1. Unique file identifier (V,I) of a file

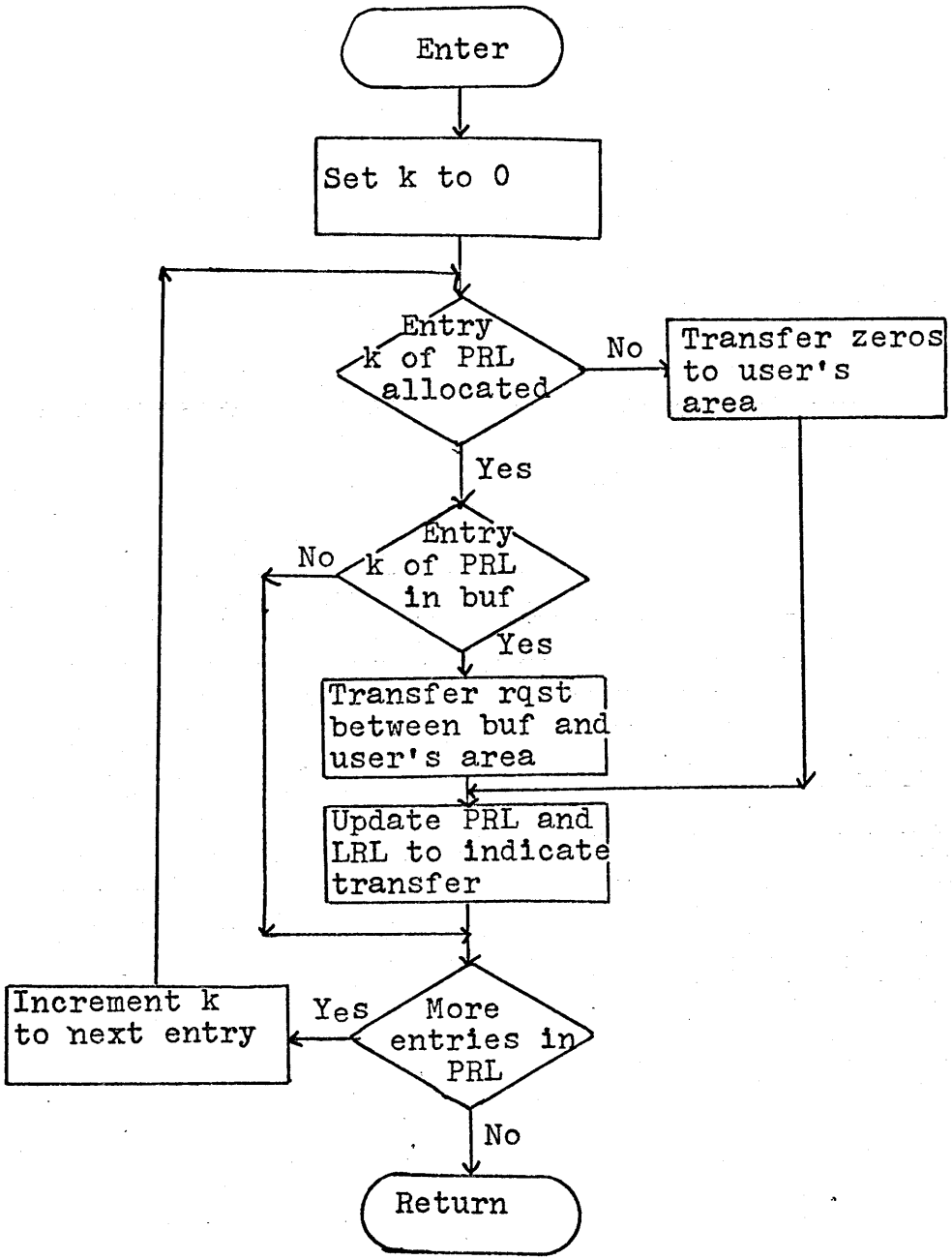


FOSM

FLOWCHART FOR ALGORITHM OF TRANSFER BUFFER TOUTINE (TRBUF)

Arguments of TRBUF:

1. Logical Record List (LRL)
2. Physical Record List (PRL)
3. Read or Write command
4. Volume (V) of the file

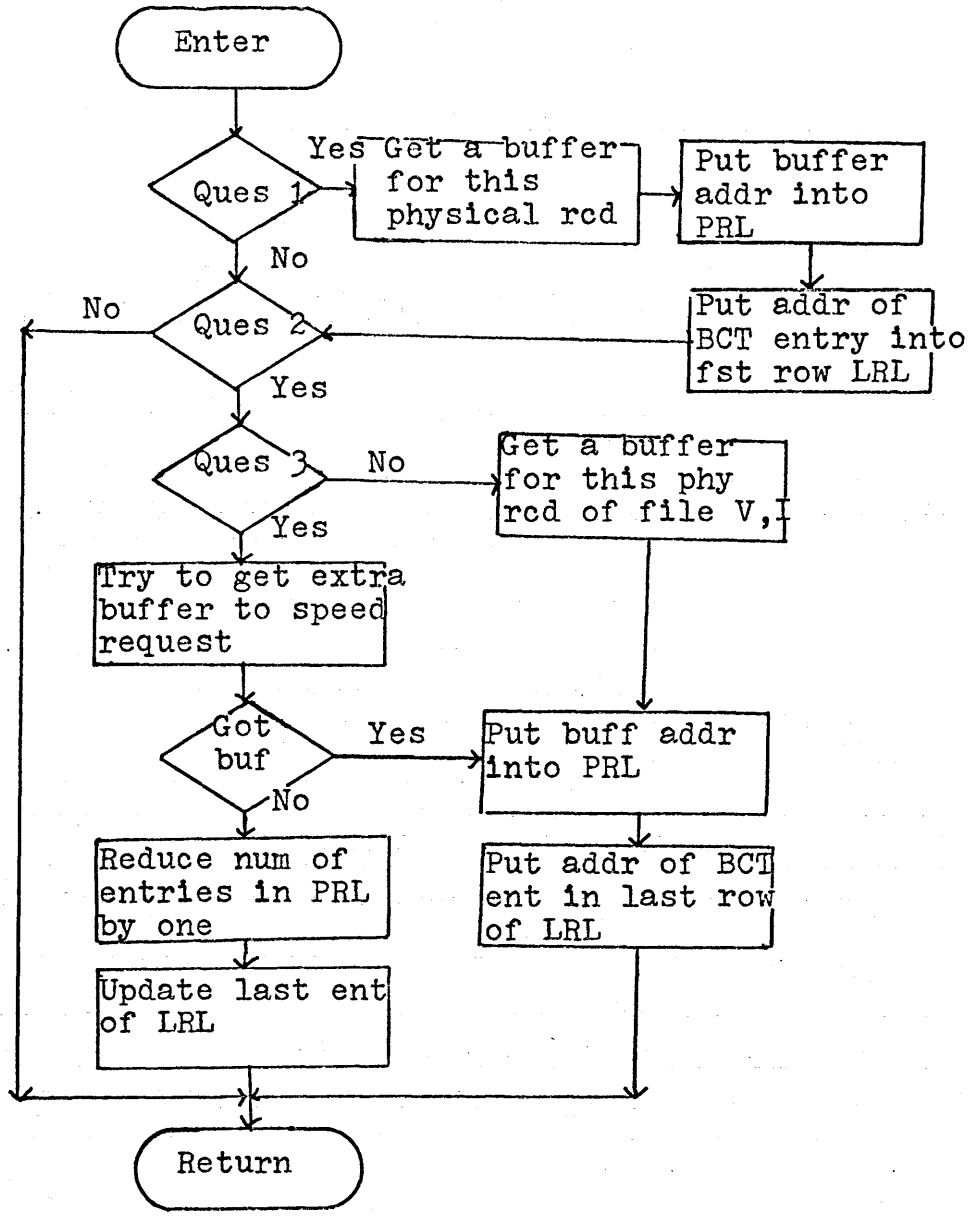


FOSM

FLOWCHART FOR ALGORITHM OF ASSEMBLE BUFFERS ROUTINE (ASBUF)

Arguments of ASBUF:

1. Logical Record List
2. Physical Record List
3. V, I, of the file



Ques 1?--Is a buffer needed for first request in Physical Record List?

Ques 2?--Is a buffer needed for last request in Physical Record List?

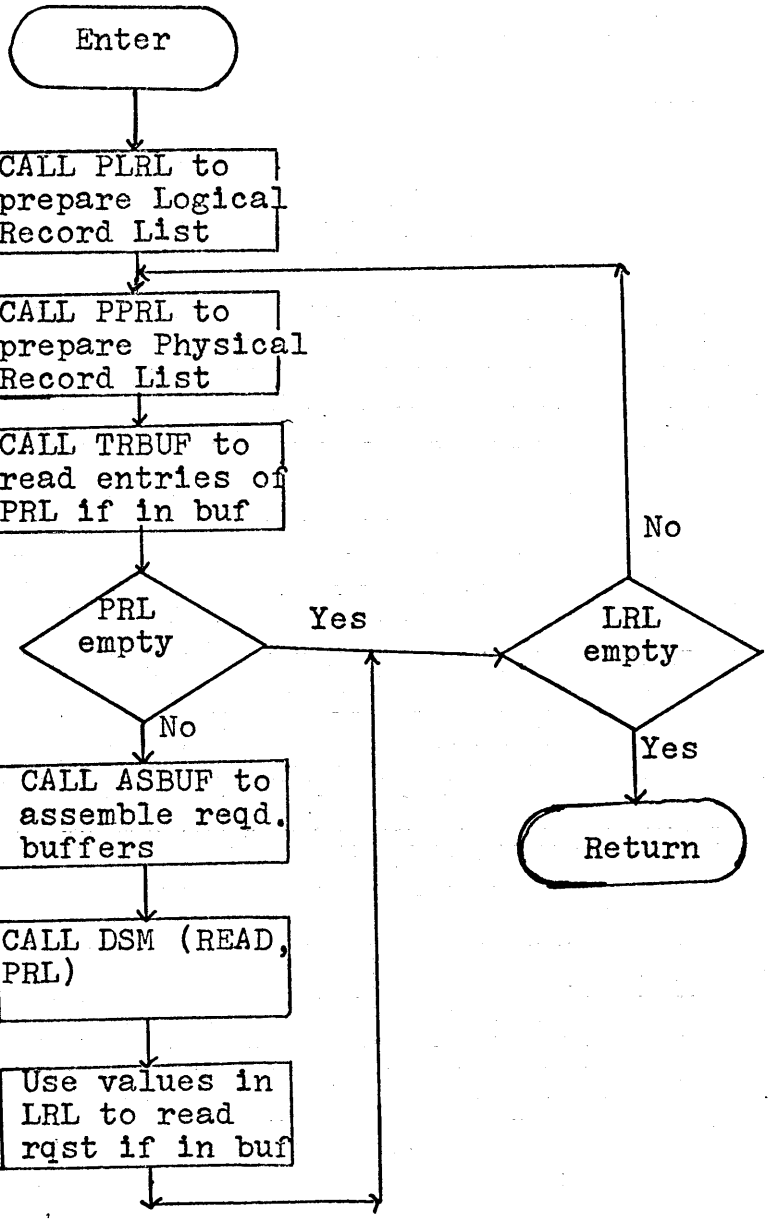
Ques 3?--Do we already have one buffer?

FOSM

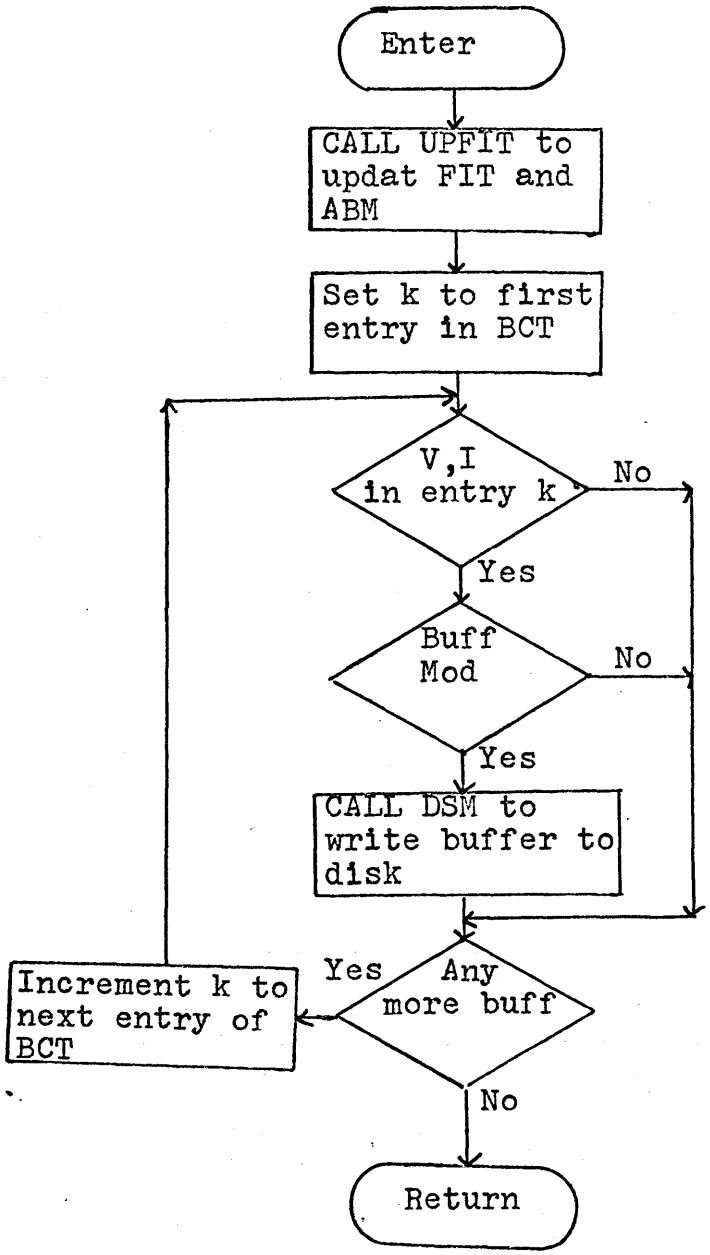
FLOWCHART FOR ALGORITHM OF READ SUBMODULE

Arguments of READ:

1. Core Address (CA)
2. File Address (FA)
3. Number (NUM) of words to transfer
4. Volume and Index (V,I) of the file
5. Physical Record Address (PRCDA) of FIT for this file



FOSM
FLOWCHART FOR ALGORITHM OF CLOSE SUBMODULE
Arguments of CLOSE:
1. Unique file identifier (V,I) of file

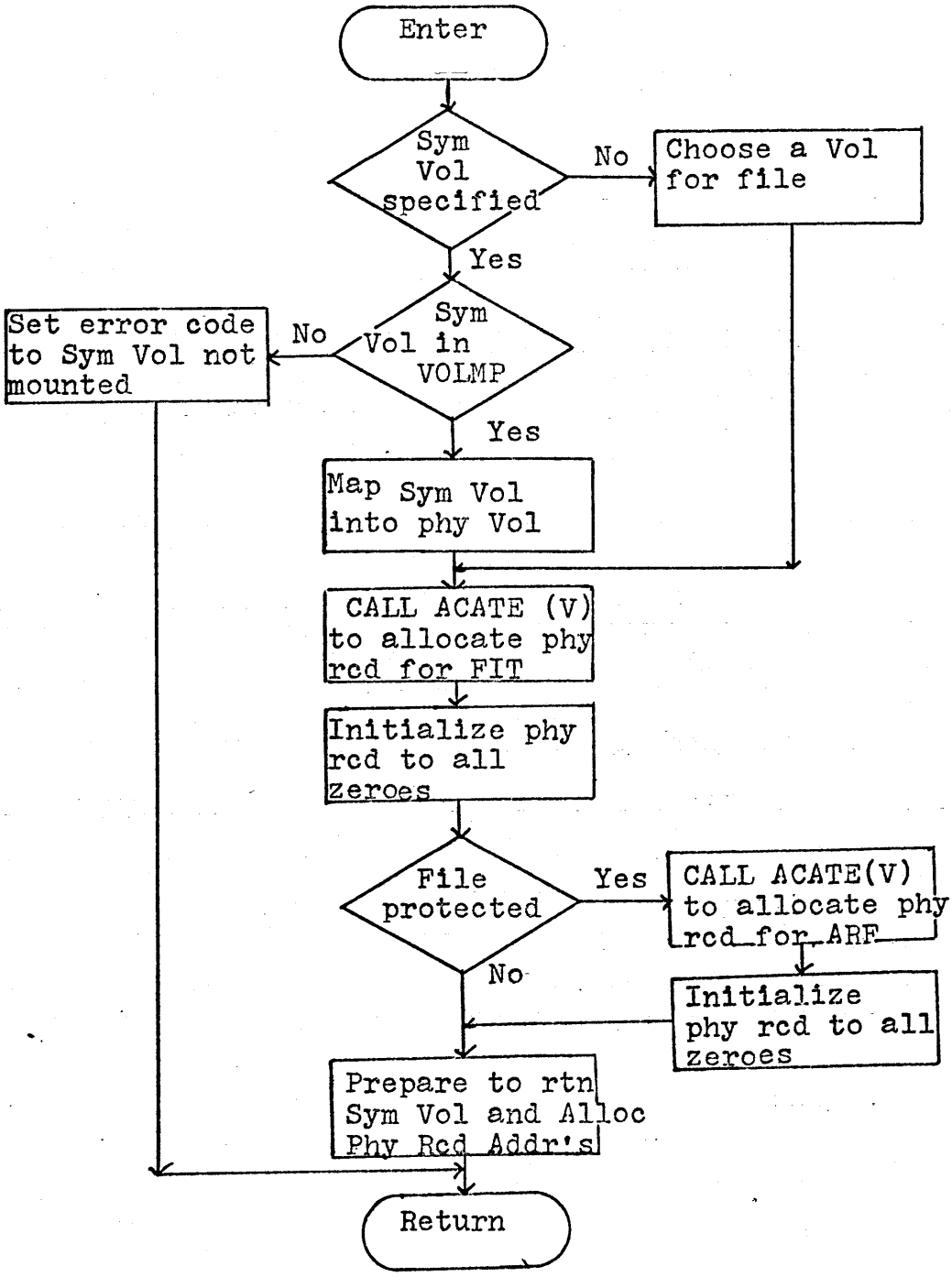


FOSM

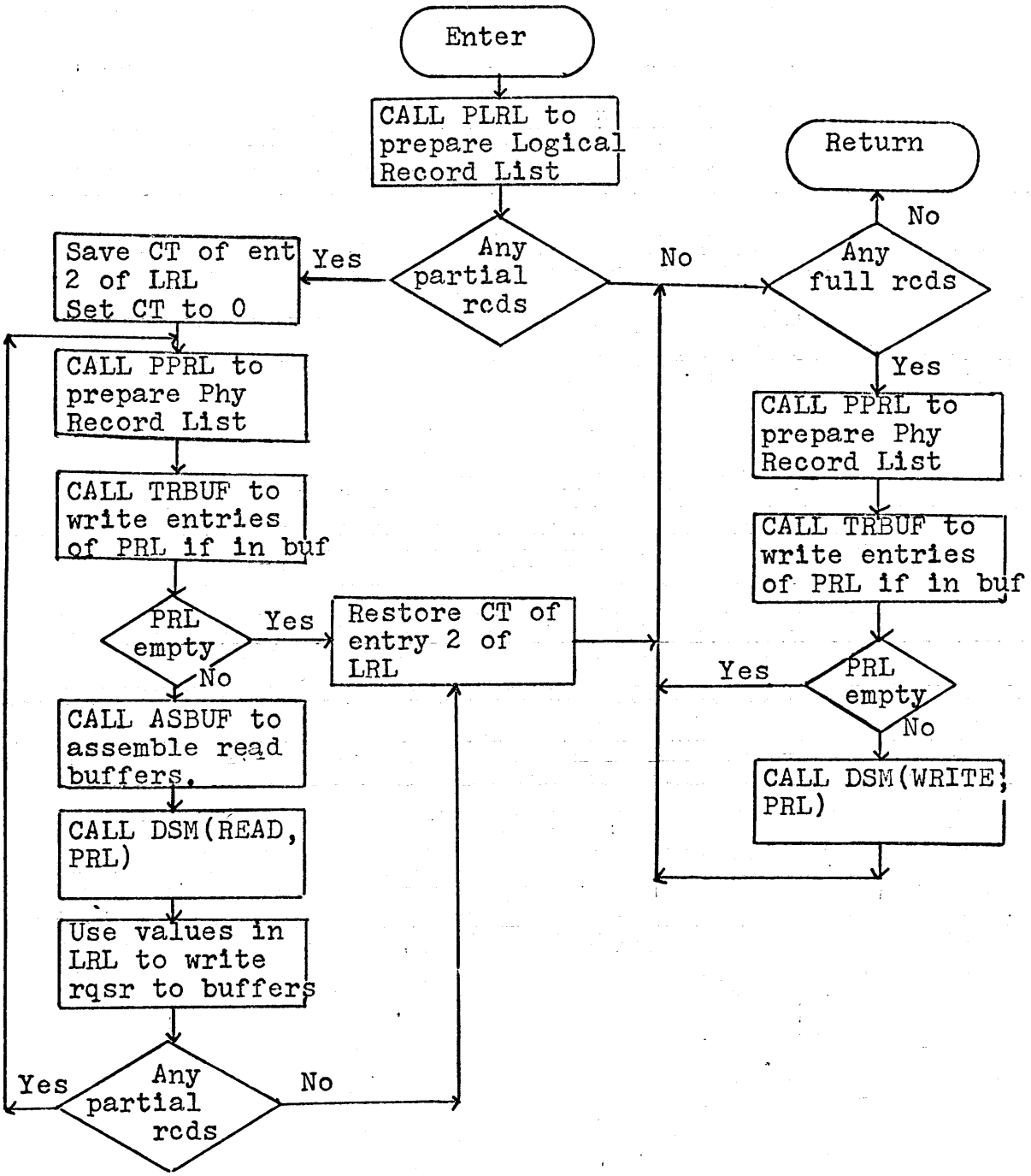
FLOWCHART FOR ALGORITHM OF CREATE SUBMODULE (CREAT)

Arguments of CREAT:

1. Symbolic Volume may be given. This is optional.
2. Access Rights (ACRTS) specified by owner



FOSM
FLOWCHART FOR ALGORITHM OF WRITE SUBMODULE
Arguments of WRITE:
1. Core Address (CA)
2. File Address (FA)
3. Number (NUM) of words to transfer
4. Volume and Index of file
5. Physical Record Address (PRCDA) of FIT

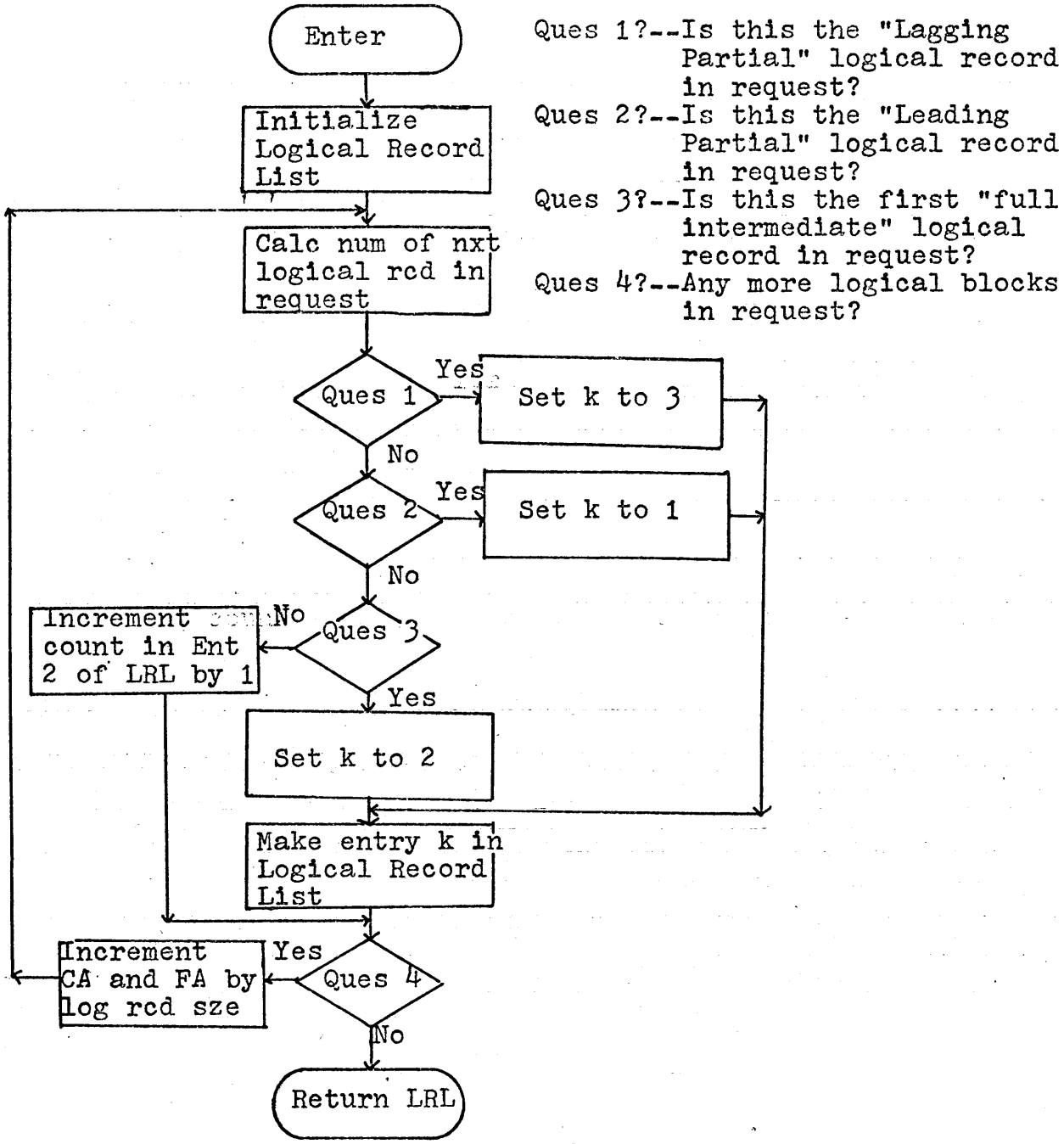


FOSM

FLOWCHART FOR ALGORITHM OF PREPARE LOGICAL RECORD LIST (PLRL)

Arguments of PLRL:

1. Core Address of request
2. File Address of request
3. Number of words in request

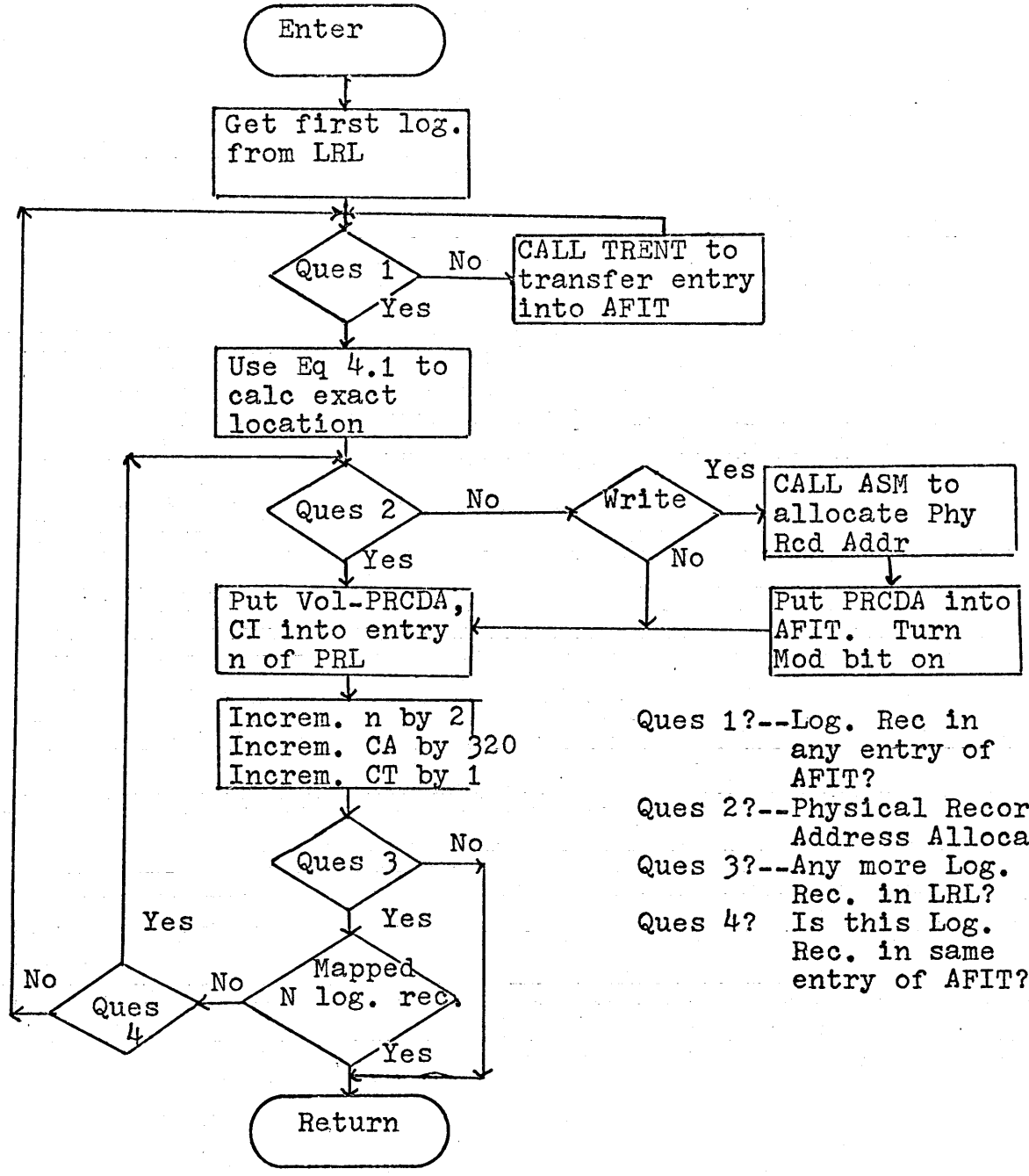


FOSM

FLOWCHART FOR ALGORITHM OF PREPARE PHYSICAL RECORD LIST ROUTINE (PPRL)

Arguments of PPRL:

1. Read or Write command
2. Logical Record List (LRL)
3. Volume (V) of file
4. Physical Record Address (PRCDA) of File Index Table

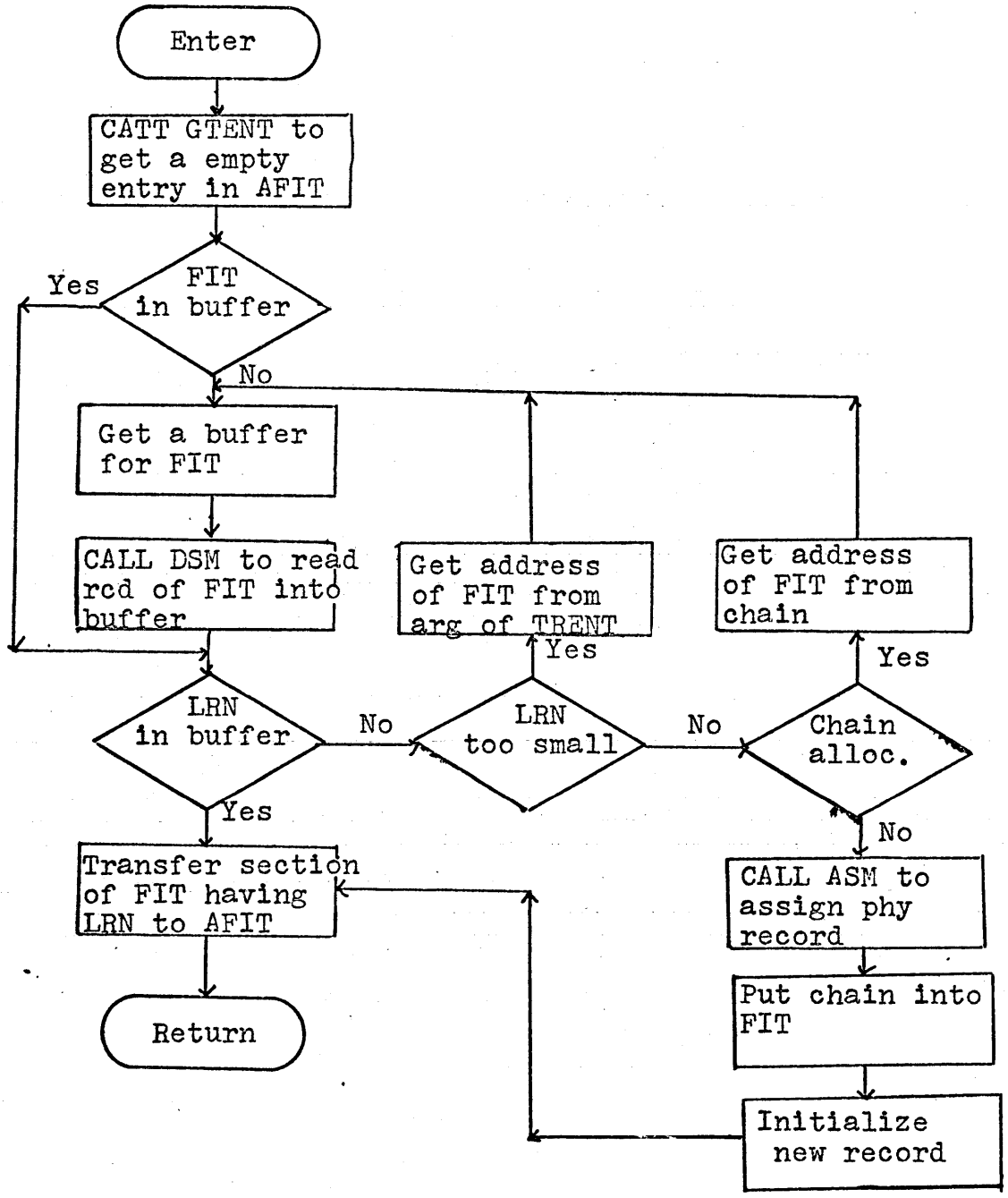


FOSM

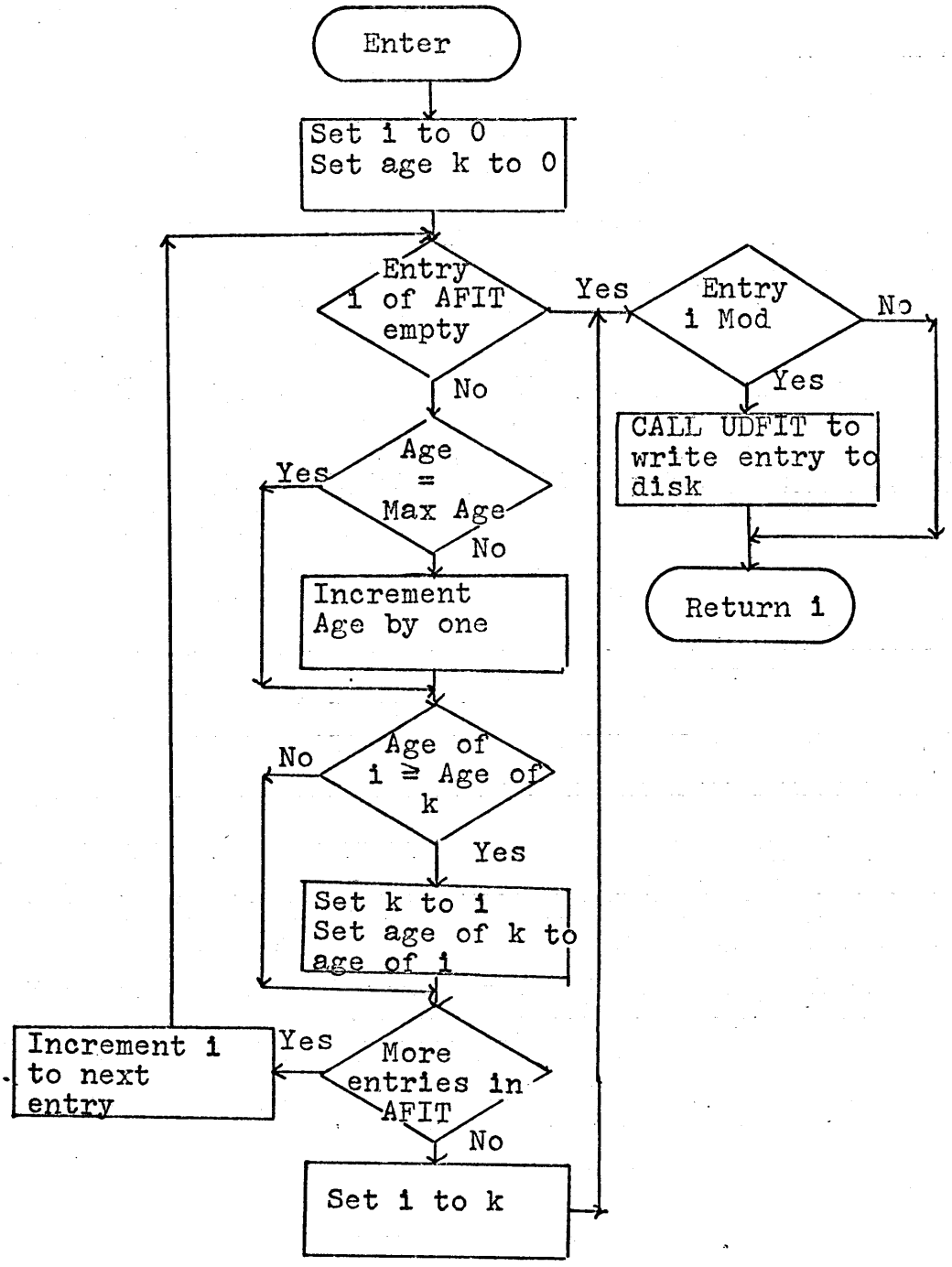
FLOWCHART FOR ALGORITHM OF TRANSFER ENTRY ROUTINE (TRENT)

Arguments of TRENT:

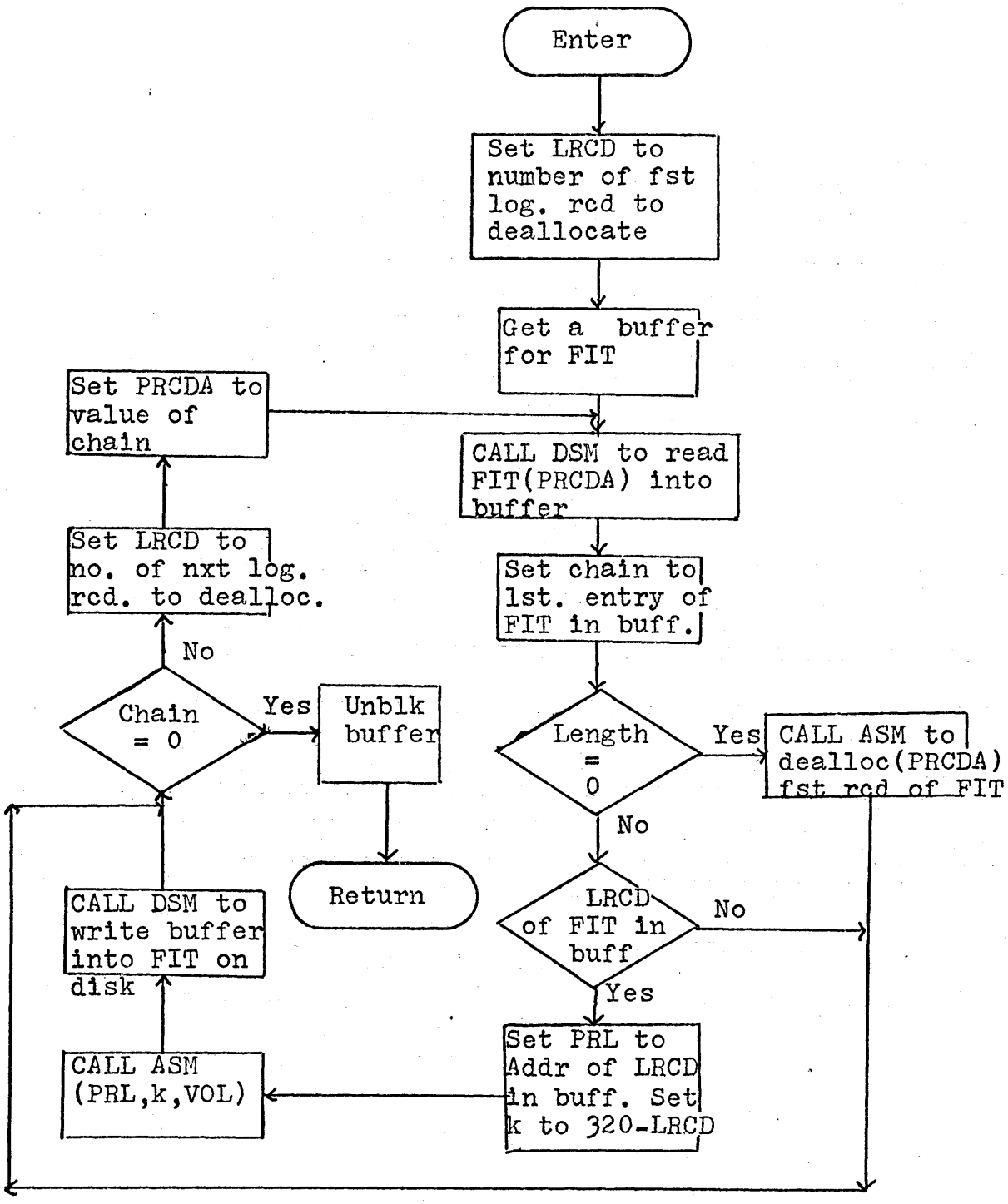
1. Volume and Index (V,I) of file
2. Physical Recore Address (PRCDA) of FIT
3. Logical Record Num (LRN) to transfer to AFIT



FOSM
FLOWCHART FOR ALGORITHM OF GET ENTRY ROUTINE
(GTENT)



FOSM
FLOWCHART FOR ALGORITHM OF TCATE SUBMODULE
Arguments of TCATE:
1. Physical Record Address (PRCDA) of FIT
2. Volume and Index of file
3. Length of file



DATA BASES OF FOSM FOR IMPLEMENTATION ON IBM 1130 COMPUTER

The discussion of these data bases is contained within Chapter IV.

File Index Table

Entry	Physical Record Address

Each entry is one word long and contains a physical record address if allocated; else, the entry contains a zero to indicate the physical record has not been allocated.

Active File Index Table

Entry of Index Part	Vol-PRCDA	SLRCD	MOD	Age	Idx

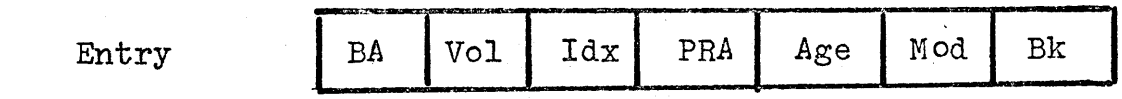
Each entry of the Index part of the Active File Index Table contains three words. The first two fields each occupy one word. The last three fields collectively occupy one word as shown below.

Bits	0	1-4	5 - 15
Function	Mod	Age	Idx

Entry of Indexed Part	PRCDA	0	PRCDA	...	0
	1	2	3	...	29

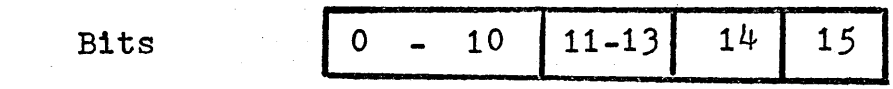
Each entry contains 29 consecutive entries of the File Index Table.

Buffer Control Table

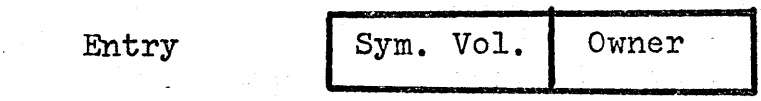


Each entry contains three words. The buffer address (BA) consists of one word. The physical volume and index representing the unique file identifier occupy one word.

The last four fields collectively occupy one word.



Symbolic Volume Map



Each entry of the Symbolic Volume Map contains two words. The symbolic volume name consists of four characters or decimal numbers in hexidecimal representation. The Symbolic Volume Map has one entry for each disk device. The index of an entry in the map is the physical address

of the disk device having the symbolic name contained within the entry. The owner field specifies if the symbolic volume mounted on the disk drive is a system volume or a personal volume. The FOSM allocates a volume for a file which is being created only on system volumes.

The FOSM also has knowledge of the location of the Active Volume Map which is a data base of the Allocation Strategy Module. The FOSM only reads the count field of the Active Volume Map to decide which system volume to allocate to a file which is being created.

APPENDIX D

This appendix contains the detailed logical flowcharts for the algorithms and the data bases for a specific design of an Allocation Strategy Module for implementation on the IBM 1130 computer.

The ASM is called by the FOSM. The allowable calls from the FOSM are listed below in flowchart notation as discussed in Appendix B.

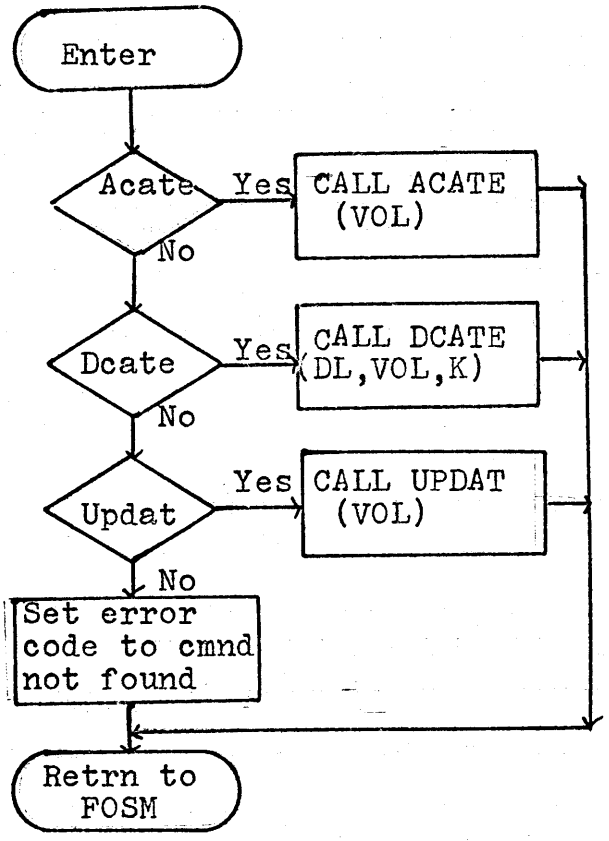
1. CALL ASM(ACATE,VOL)
2. CALL ASM(DCATE,DL,VOL,K)
3. CALL ASM(UPDAT,VOL)

ASM

FLOWCHART FOR ALGORITHM OF MAINLINE MODULE (ASM)

Arguments of ASM:

1. See preceeding page

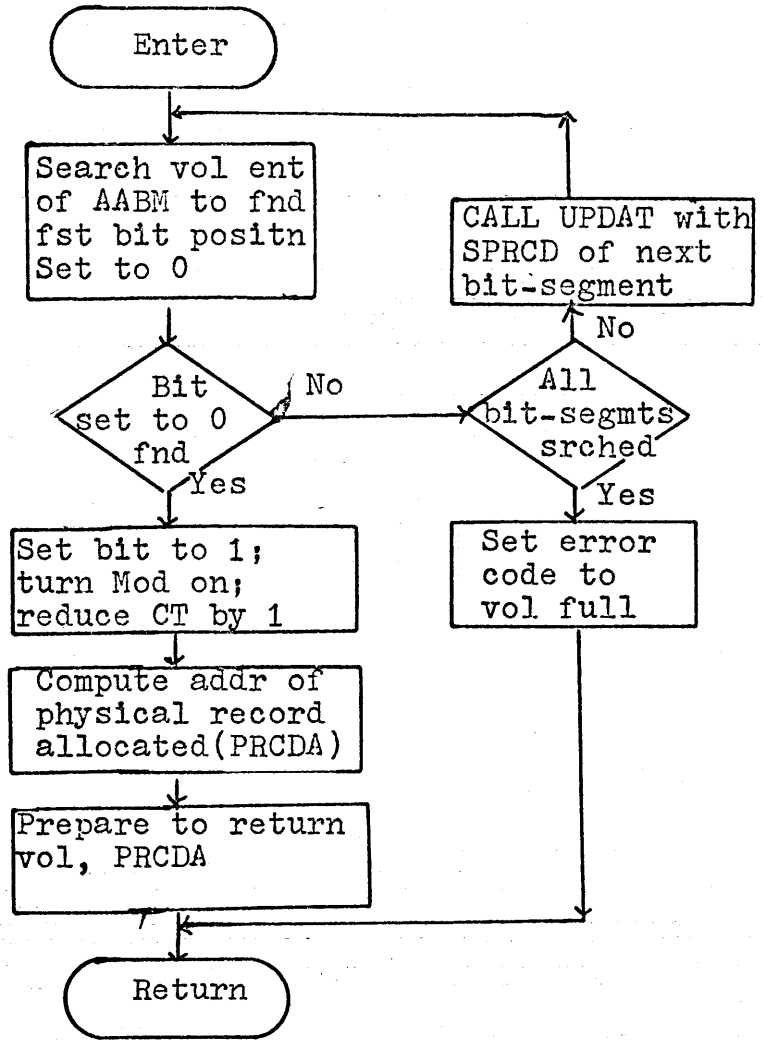


ASM

FLOWCHART FOR ALGORITHM OF ALLOCATE SUBMODULE (ACATE)

Arguments of ACATE:

1. Volume assigned to a file

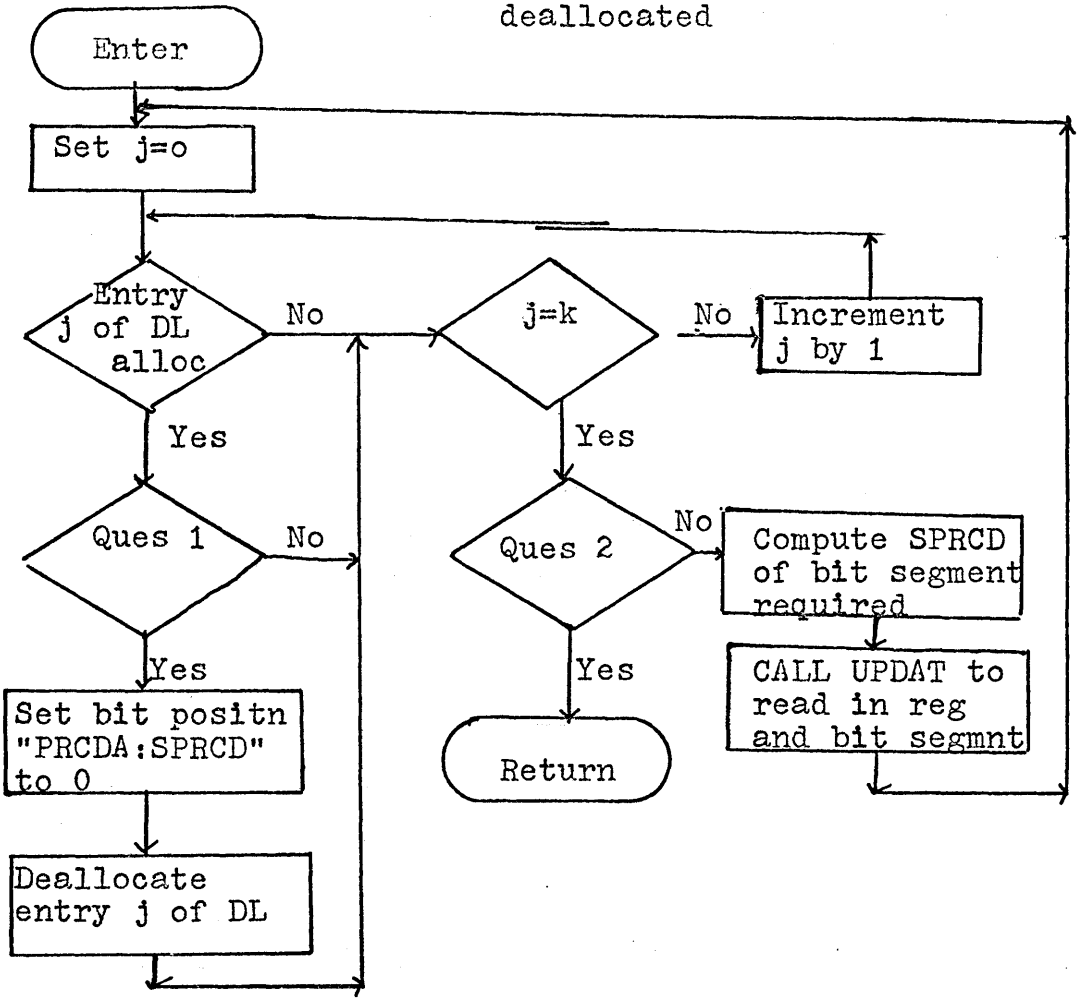


ASM

FLOWCHART FOR ALGORITHM OF DEALLOCATE SUBMODULE

Arguments of DCATE:

1. Deallocation list (DL) containing physical record addresses to be deallocated
2. The Number, k, of entries in the PRCDL
3. Volume containing physical records to be deallocated



Question 1?--Is bit position, corresponding to the physical record of entry j of DL, currently in AABM? Equivalently, is SPRCD $\leq$ PRCDA? Equivalently is SPRCD$\leq$PRCDA$\leq$SPRCD+16*20?

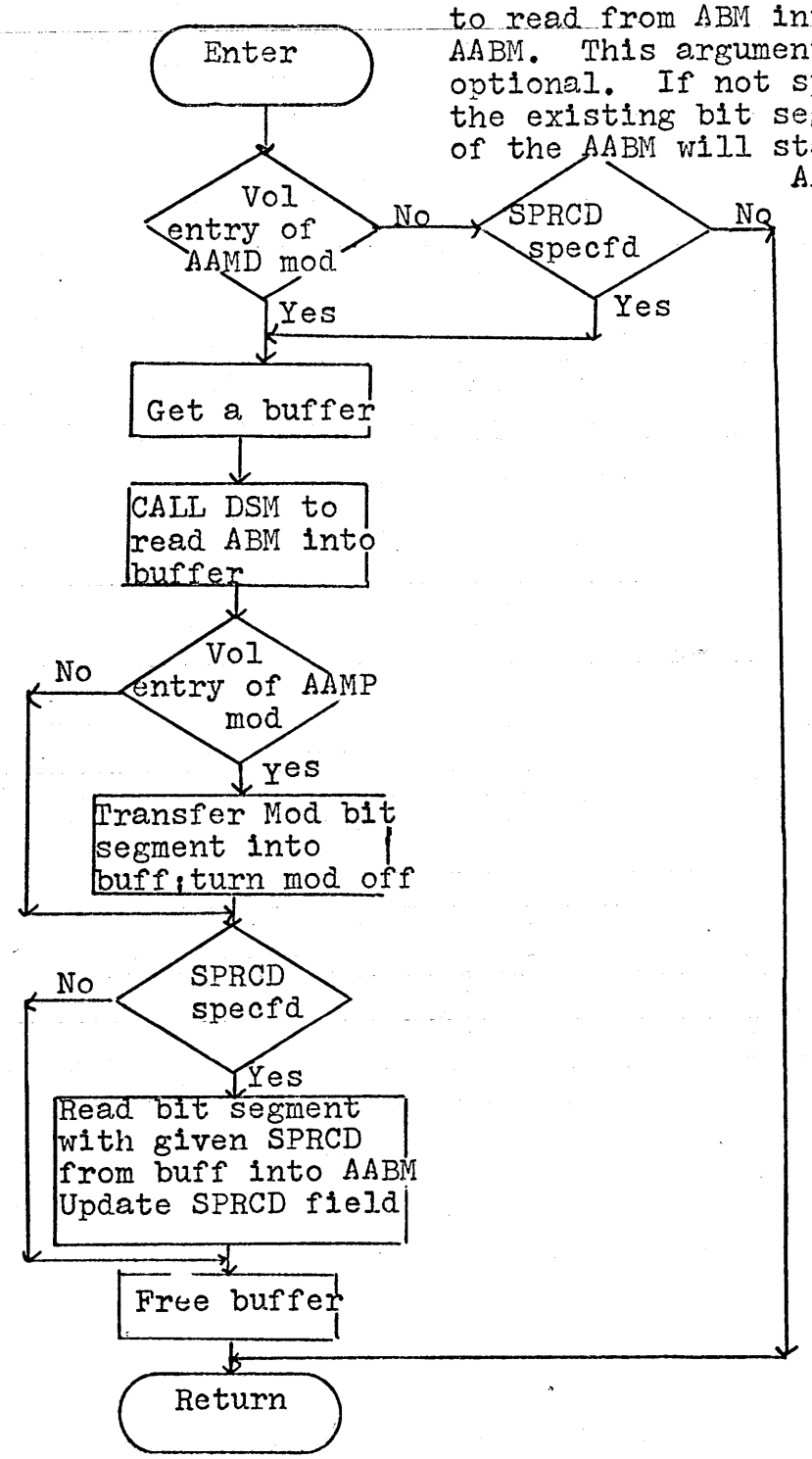
Question 2?--Have all entries of DL been deallocated?

ASM

FLOWCHART OF ALGORITHM OF UPDATE UTILITY ROUTINE

Arguments of UPDAT:

1. Volume assigned to file
2. Starting Physical Record (SPRCT) of bit segment to read from ABM into AABM. This argument is optional. If not specified, the existing bit segment of the AABM will stay in AABM.



DATA BASES FOR IMPLEMENTATION ON IBM 1130 COMPUTER

Active Allocation Bit Map

Entry	21-word bit segment	SPRCD	PRCDA	CT	Mod

There is one 25-word entry in the AABM for each mounted volume. The SPRCD, PRCDA, CT and MOD fields each occupy one word in the entry.

Allocation Bit Map

The ABM consists of five contiguous 21-word bit segments representing 1680 physical records. Since a disk volume for the IBM 1130 has only 1624 physical records available, bit positions 1625-1680 will be turned on initially to give the illusion that they have been allocated.

APPENDIX E

This appendix contains the logical flowcharts for the algorithms for a specific design of the Device Strategy Module and the interrupt processing routines for implementation on an IBM 1130 computer. Each disk device requires an individual interrupt processing routine. Since they are all alike, the flowcharts of only one set of the interrupt processing routines are given in this appendix.

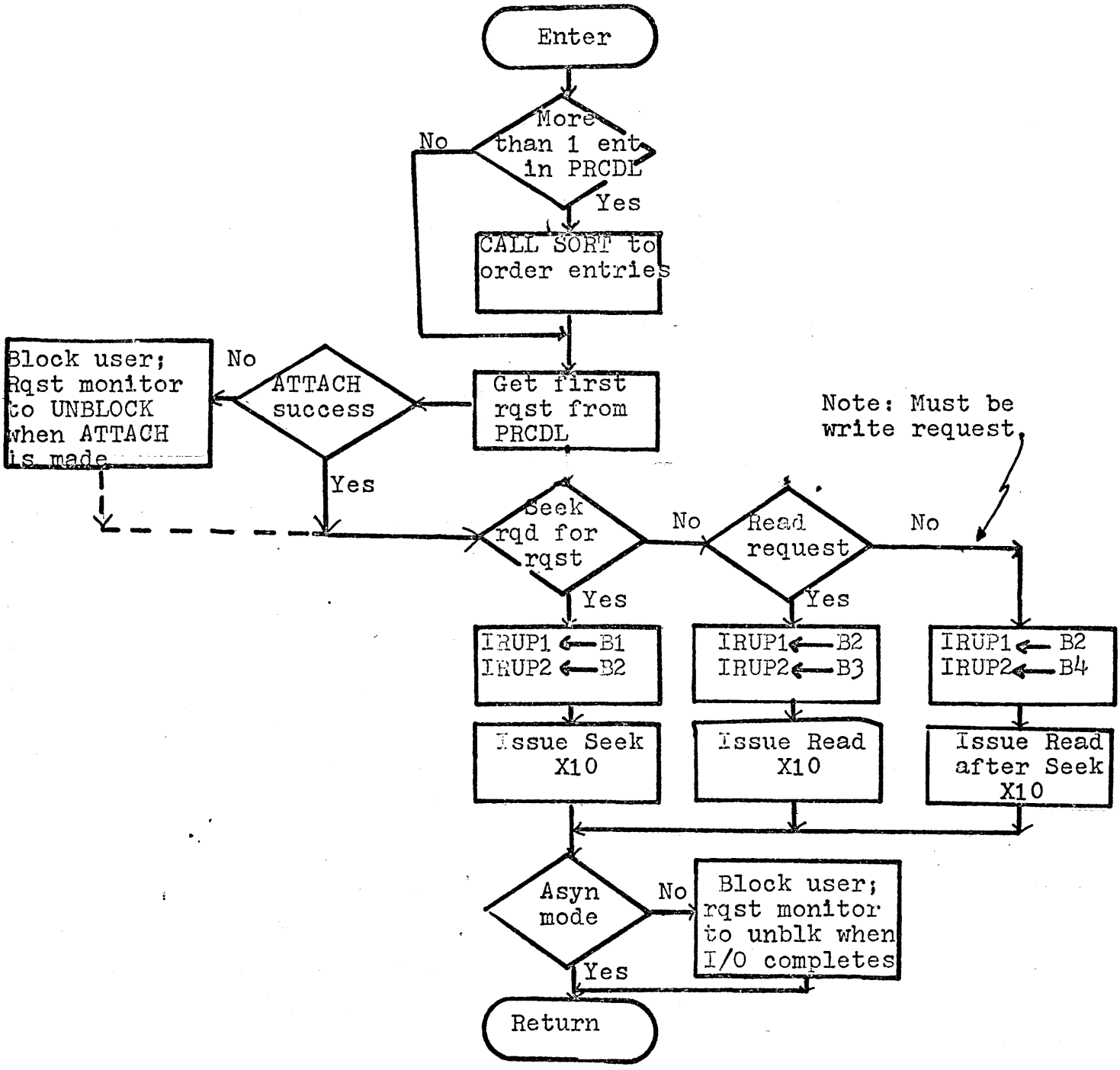
The DSM is called by the FOSM and ASM. The flowchart notation for the single allowable call is given below.

1. CALL DSM(PRCDL,MODE)

Arguments of DSM call:

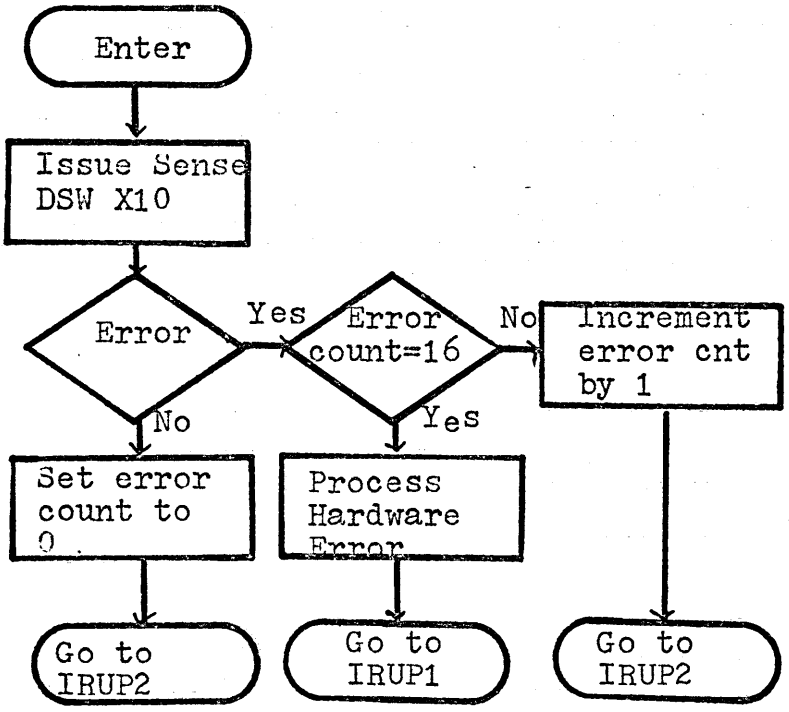
1. The physical record list (PRCDL) having the format discussed in the FOSM.
2. The Mode of operation. The two allowable modes are synchronous and asynchronous.

DSM
FLOWCHART FOR ALGORITHM OF DEVICE STRATEGY MODULE
1. Arguments of DSM: see previous page

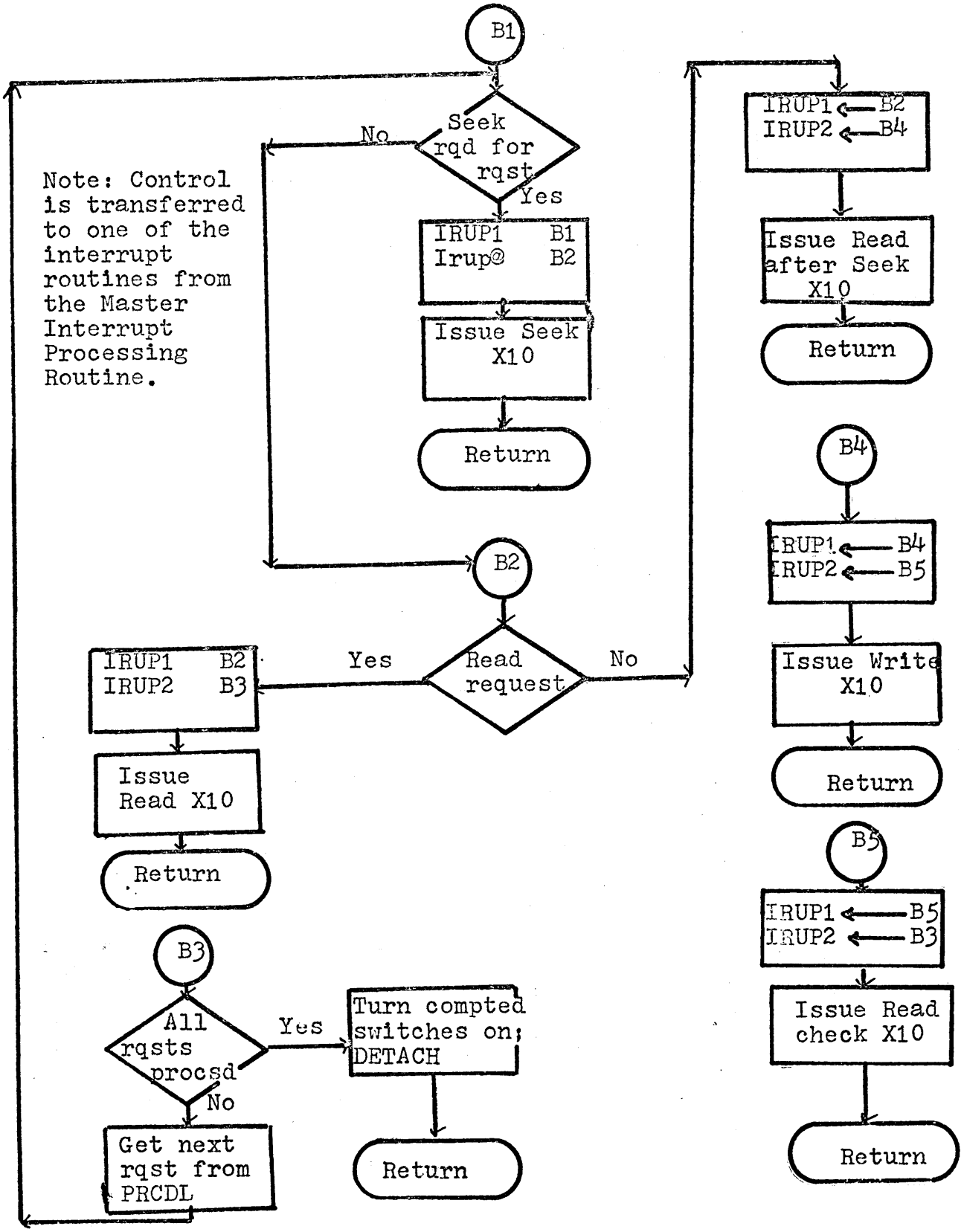


DSM
FLOWCHART FOR ALGORITHM OF MASTER INTERRUPT PROCESSING ROUTINES FOR ONE DISK DEVICE

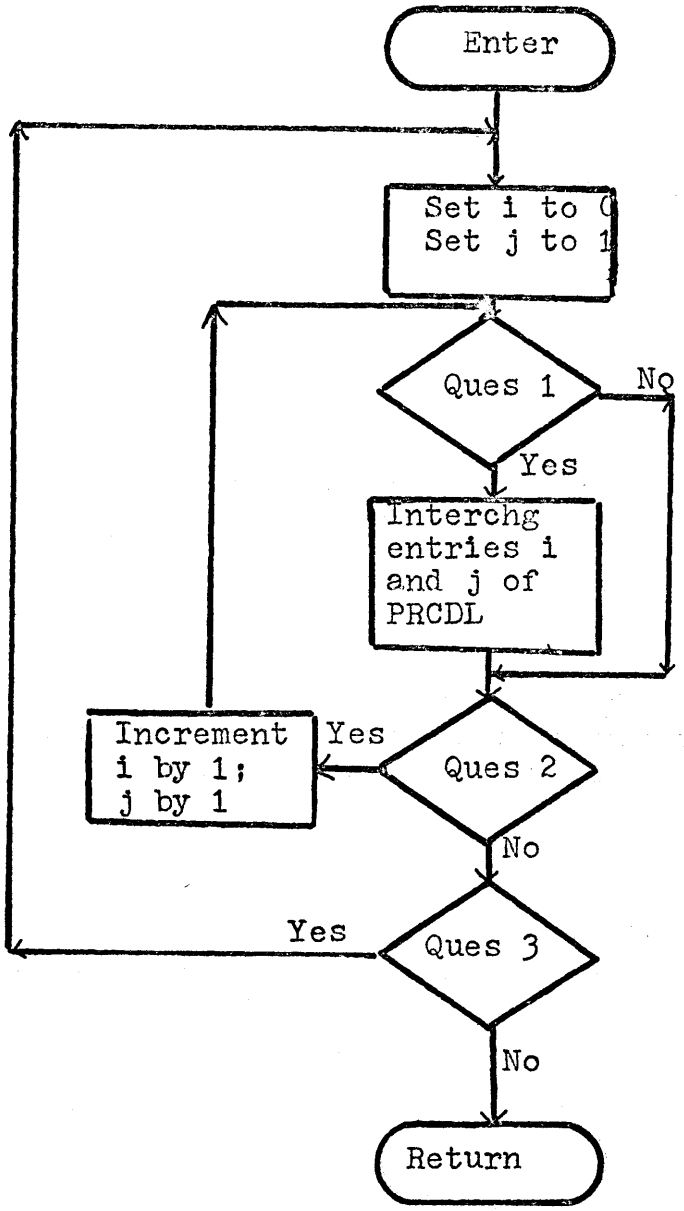
Note: Control from I/O Controller enters here.



DSM
FLOWCHART FOR ALGORITHM OF SECONDARY INTERRUPT PROCESSING ROUTINES FOR ONE DISK DEVICE



DSM
FLOWCHART FOR ALGORITHM OF SORT ROUTINE
Arguments of SORT:
1. Physical Record List (PRCDL)



Note: A simple interchange sort was chosen since the number of requests in the PRCDL is expected to be small in the 1130 environment.

Question 1. Is physical record number of entry i in PRCDL greater than physical record number of entry j?
Question 2. Any more entries in PRCDL?
Question.3. Did we interchange any entries the last time through?

REFERENCES

1. Abraham, C.T., Ghosh, S.P., and Ray-Chaudhuri, D.K., File Organization Schemes Based on Finite Geometries, Information and Control, February, 1968.

2. Bash, J.L., Benjafield, E.G., and Gondy, M.L., The MULTICS Operating System, Cambridge Information Systems Laboratory, May, 1967.

3. Daley, R.C., and Neumann, P.G., A General Purpose File System for Secondary Storage, Proceedings Fall Joint Computer Conference, 1965.

4. Denning, P.J., Queuing Models for File Memory Operations, MIT Project MAC Technical Report MAC-TR-21, October, 1965.

5. Dennis, J.B., Segmentation and the Design of Multi-Programmed Computer Systems, Journal of ACM, October, 1965.

6. Dixon, P.J., and Sable, D.J., DM-1--A Generalized Data Management System, Proceedings Spring Joint Computer Conference, 1967.

7. Evans, D.C., and LeClerc, J.V., Address Mapping and the Control of Access in an Interactive Computer, MIT Project MAC Document Room, December, 1966.

8. Graham, R.M., Protection in an Information Processing Utility, Communications of the ACM, May, 1968.

9. Hartley, D.F., Landy, B., and Needham, R.M., The Structure of a Multiprogramming Supervisor, Computer j.,

November, 1968.

10. Henry, W.R., Hierarchical Structure for Data Management, IBM Systems Journal, Vol. 8, No. 1, 1969.

11. Hollander, C.R., A Multi-Tasking Monitor for the IBM 1130 Computer, MIT Department of Electrical Engineering, June, 1969.

12. Madnick, S.E., Design Strategies for File Systems, S.M. thesis, MIT Department of Electrical Engineering, June, 1969.

13. Rappaport, R.L., Implementing Multi-process Primitives in a Multiplexed Computer System, S.M. thesis, MIT Department of Electrical Engineering, August, 1968.

14. Rosen, Saul, Programming Systems and Languages, McGraw-Hill, New York, 1967.

15. Saltzer, J.H., Traffic Control in a Multiplexed Computer System, Sc.D. thesis, MIT Department of Electrical Engineering, August, 1968.

16. Zilles, S.N., Synchronization of Resource Usage in a Small Information System